TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

OLLI-PEKKA HEINISUO

DECENTRALIZED FILE SHARING ON MOBILE DEVICES

Master's thesis

Examiners: Asst. Prof. Davide Taibi and Postdoctoral Researcher Valentina Lenarduzzi

The examiners and topic of the thesis were approved on 29 August 2018

ABSTRACT

OLLI-PEKKA HEINISUO: Decentralized File Sharing on Mobile Devices
Tampere University of Technology
Master of Science Thesis, 52 pages
November 2018
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiners: Asst. Prof. Davide Taibi and Postdoctoral Researcher Valentina Lenarduzzi

Keywords: decentralization, mobile application, p2p, file sharing

Most applications and services rely on central authorities. This introduces a single point of failure to the system. The central authority must be trusted to have data stored by the application available at any given time. More importantly, the privacy of the user depends on the service provider capability to keep the data safe. To remove the central authority a decentralized system is needed. Due to the rapid growth of mobile device usage, the availability of decentralization must not be limited only to desktop computers.

This thesis examined the possibility to use mobile devices as a decentralized file sharing platform without any central authorities. This was done by implementing a peer-to-peer file sharing mobile application. InterPlanetary File System was selected as the peer-to-peer network. To evaluate the validity of the approach, network usage and power consumption of multiple different devices was measured while the application was running.

The results indicate that mobile devices can be used to implement a worldwide distributed file sharing network. However, the file sharing application generated large amounts of network traffic even when no files were shared. This was caused by the chattiness of the protocol of the underlying peer-to-peer network. Consequently, constant network traffic prevented the mobile devices from entering to deep sleep mode. Due to this the battery life of the devices was greatly degraded. Further measurements are needed when the InterPlanetary File System reference implementation is more mature. A paper about the results of this thesis has been submitted to the Seventh IEEE International Conference on Mobile Cloud Computing [36].

TIIVISTELMÄ

OLLI-PEKKA HEINISUO: Hajautettu tiedostojenjako mobiililaitteilla
Tampereen teknillinen yliopisto
Diplomityö, 52 sivua
Marraskuu 2018
Tietotekniikan DI-tutkinto-ohjelma
Pääaine: Pervasive Systems
Tarkastajat: apulaisprofessori Davide Taibi ja tutkijatohtori Valentina Lenarduzzi

Avainsanat: hajautettu järjestelmä, mobiilisovellus, vertaisverkko, tiedostonjako

Useimmat sovellukset ja palvelut ovat riippuvaisia keskitetyistä järjestelmistä. Sovellusten käyttäjien on luotettava siihen, että palvelut ovat saatavilla milloin tahansa. Tämän lisäksi käyttäjien tietojen turvallisuus on kokonaan palveluntarjoajan käsissä. Tarve keskitetyille palveluille voidaan kiertää täysin hajautetuilla järjestelmillä. Mobiililaitteiden käytön suuren kasvun vuoksi tällaista hajautettua järjestelmää ei voida rajoittaa pelkästään työpöytäkäyttöön.

Tässä työssä tutkittiin, onko mobiililaitteita mahdollista käyttää täysin hajautetun tiedostojenjakopalvelun alustana. Tätä varten työssä toteutettiin vertaisverkkoon perustuva tiedostojenjakosovellus mobiililaitteille. Vertaisverkoksi valittiin InterPlanetary File System. Toteutuksen käyttökelpoisuuden arvioimiseksi usean eri mobiililaitteen virrankulutus sekä verkon käyttö mitattiin sovelluksen ollessa päällä.

Tulosten perusteella mobiililaitteita voidaan käyttää alustana maailmanlaajuisen hajautetun tiedostonjakoverkon toteuttamiseen. Tästä huolimatta on huomioitava, että kehitetty sovellus käytti verkkoa todella paljon. Tämä johtui käytetyn vertaisverkon protokollasta, joka vaihtoi tietoa muun verkon kanssa jatkuvasti. Jatkuvasta verkkoliikenteestä johtuen mitatut laitteet eivät koskaan päässeet lepotilaan, mikä puolestaan johti suureen virrankulutukseen. Tehdyt mittaukset tulisi toistaa, kun käytetyn vertaisverkon referenssitoteutusta on optimoitu ja kehitetty edelleen. Työn tuloksista on lähetetty tieteellinen artikkeli Seventh IEEE International Conference on Mobile Cloud Computing –konferenssiin [36].

PREFACE

This thesis is the end result of working and studying at the same time for the past seven and a half years. It was not easy to figure out a good topic but I am happy that I ended up with this one. While this thesis is my personal work and it was done without external support, I am grateful for the wide industry experience I have gained at the companies at which I have worked during the past years.

I would like to thank Assistant Professor Davide Taibi for the feedback during the writing process of this thesis and for the opportunity to summarize the work into a scientific paper. I would also like to express my gratitude to my family and friends. Special thanks to my girlfriend for supporting me during the thesis work.

In Tampere, Finland, on 15th November 2018

Olli-Pekka Heinisuo

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

ARM	an processor architecture
ARPANET	Advanced Research Projects Agency Network, the early Internet
CI	Continuous Integration
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DHT	Distributed Hash Table
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ID	Identifier
IPFS	InterPlanetary File System
IPLD	InterPlanetary Linked Data
IPNS	InterPlanetary Naming System
MFS	Mutable File System
NAT	Network Access Translation
OS	Operating System
p2p	peer-to-peer
QML	Qt Modeling Language
Qt	cross-platform application development framework
RPM	Red Hat Package Manager
SDK	Software Development Kit
SFS	Self-certified File System
SSH	Secure Shell
TCP/IP	Transmission Control Protocol / Internet Protocol
TTL	Time-To-Live
UDP	User Datagram Protocol
UI	User Interface
WLAN	Wireless Local Area Network
x86	an processor architecture

1. INTRODUCTION

The usage of mobile devices has increased significantly during the past 10 years. In 2016, the web was accessed for the first time more via mobile devices than traditional computers [61]. The users of all these billions of network connected devices are producing and consuming more data than ever before. To make the data accessible to other users and devices it has to be shared over the network.

Many of the mobile devices are connected to one or multiple services which are often running on some cloud platform. Some examples of the most used cloud platforms are Amazon's AWS [3] and Microsoft's Azure [4]. These platforms provide computing resources, for example storage space and processing power, to build scalable web-based software services. To the end users cloud services are usually completely transparent: they make it possible for software applications to share data between different devices everywhere in the world.

Cloud usage has been growing rapidly in recent years and it will continue to grow in the future [12]. Consequently, more and more personal data of the end users is stored in large centralized data centers which are owned and controlled by a few large private enterprises [17]. While centralized storage is proven to work well, it has many implications regarding privacy, security and control of the data. Facebook's and Cambridge Analytica's illegal usage of 50 million user profiles is a recent example of these issues [57].

To avoid the issues of centralization, data can be stored without any central authorities. Decentralized storage and content distribution can be achieved with peer-to-peer (p2p) networking. In peer-to-peer networking every node of a network is both a client and a server [59]. The nodes then talk to each other without any central service as opposed to centralized systems where all communication goes through dedicated central servers.

Centralized systems depend heavily on the availability of the central servers. Often the client software will stop working properly if the servers are not available. In decentralized systems there is no single point of failure which makes them more fault tolerant than centralized systems. Centralized systems usually require also some fees to cover the infrastructure expenses. Decentralization is more cost-effective as there is no need to pay for the infrastructure: nodes participate in the upkeep of the system by sharing resources such as network bandwidth with other nodes.

Peer-to-peer networks and file sharing are rather common on desktop environments but they have been rarely used on mobile devices due to more restricted resources. However, the growth of the mobile ecosystems has changed the situation: the processing power of mobile

devices is getting better every year [34]. Similarly, the mobile network speeds are growing fast [27]. While mobile devices are used a lot, a considerable amount of processing power and network bandwidth remains still unused. Additionally, mobile devices are almost always on and connected to the Internet. These unused resources could be used to power a worldwide peer-to-peer data sharing network. Furthermore, the need for peer-to-peer content sharing mobile applications has been recognized in a user study already in 2007 [48].

This thesis examines the possibility to transform mobile devices into fully decentralized content delivery network without any central servers or authorities. This is done by implementing a file sharing mobile application which utilizes a peer-to-peer network under the hood. Apart from the application, it is measured how the peer-to-peer software affects to the performance of the mobile device to be able to better evaluate the validity of the solution. The results of this work have been submitted to the Seventh IEEE International Conference on Mobile Cloud Computing [36].

The organization of this thesis is as follows: Chapter 2 introduces the basic theory behind peer-to-peer networking. Chapter 3 discusses about the use cases and goals for the file sharing application. Chapter 4 describes the application implemented within this thesis. Chapter 5 presents the measurement results and discusses about them. Finally, Chapter 6 contains conclusions and ideas for further development.

2. THEORETICAL BACKGROUND

This chapter discusses fundamental topics about peer-to-peer networking. There exists a plethora of different peer-to-peer networks and technologies. The application implemented in this thesis is related specifically on peer-to-peer file sharing. Therefore, this chapter focuses on topics which are important in order to understand the concepts of peer-to-peer networking and file sharing.

In the first Section 2.1, basic background of different types of networks is introduced. The second Section 2.2 focuses on peer-to-peer network architectures. The third Section 2.3 gives a brief overview of the peer-to-peer networking evolution. After that Section 2.4 will go through privacy, security and trust in peer-to-peer networks. Section 2.5 discusses about how mobile environment affects to peer-to-peer applications. Section 2.6 compares modern peer-to-peer file sharing systems. Section 2.7 introduces the selected file sharing system in more depth. The final Section 2.8 reviews previous related work.

2.1 Network Architectures

The Internet has become the backbone of the modern society. Every day, more and more devices are being connected to it to be able share data and communicate more efficiently. The communication networks between different devices and services can be implemented in multiple different ways. These network architectures can be roughly split into two different types: centralized and distributed networks. The Figure 2.1 illustrates one possible approach to categorize different network models.

The network model illustration in Figure 2.1 was created by Paul Baran in the 1960's [5]. Nowadays, depending on context, the terms decentralized and distributed might be used to describe different things and sometimes they are used interchangeably. Both models B and C could be also characterized as distributed networks. Therefore, in this thesis the term decentralization refers to models where there is no central point of control or authority.

One of the very first versions of the Internet, ARPANET, was fundamentally a distributed system. As the Internet evolved, the original distributed network was slowly replaced by a more centralized structure.

Commonly, the Internet connected devices run applications which are implemented by using client-server architecture [46, p. 72]. The servers provide different services for the clients. The clients, for example mobile or desktop applications, make requests to one or more remote servers to access the services provided by the servers. The client-server architecture is a centralized architecture as illustrated in Figure 2.1 and it depends heavily

on the servers to be available at any given time. As the client amount increases, server has to handle more requests. Consequently, server resource usage increases and the server might become overloaded. To prevent overloading, the amount of servers has to be increased or server has to be upgraded to one with more processing power and other resources.

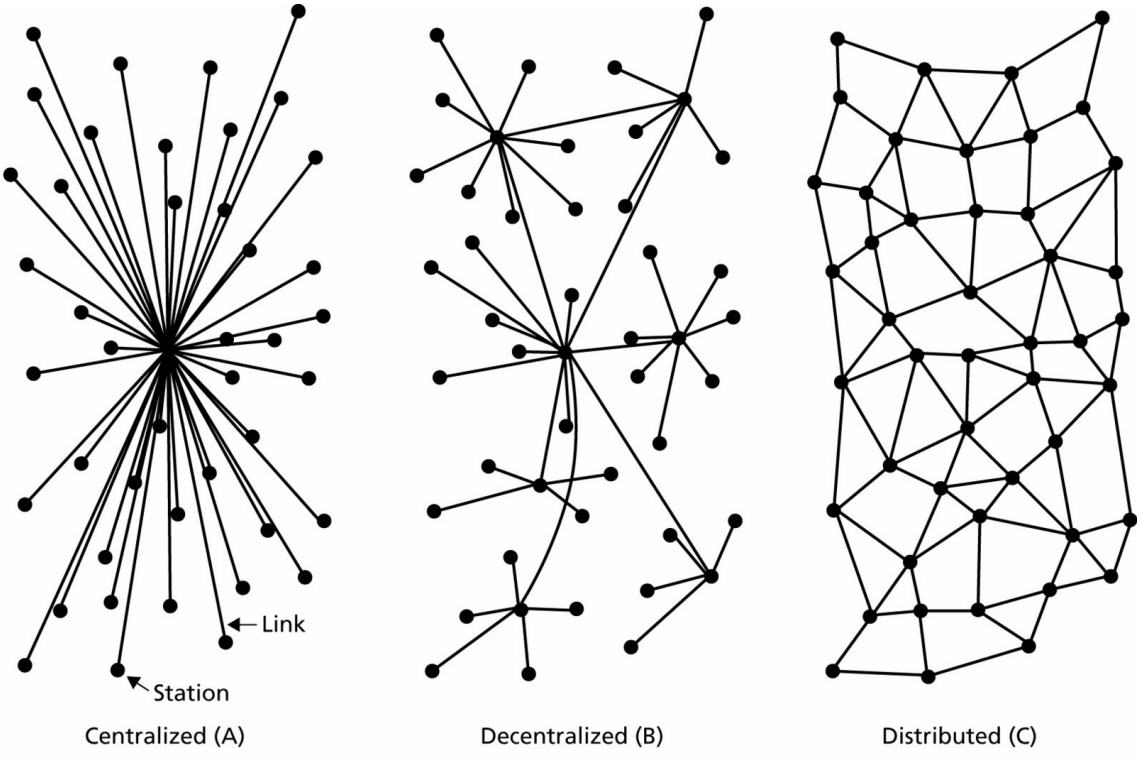

Figure 2.1. *Differences between different network models. [5]*

Due to the rise of cloud computing, physical on-premises servers have become rare. Cloud services allow to scale resources up easily or even automatically and thus make the scaling process relatively fast. This is often referred as serverless computing [21]. However, hiding the servers from the users does not make the fundamental issue go away: the servers are still there and suffer from the very same centralization issues as before.

Scaling resources up costs money. More importantly, there still exists single point of failure. The servers might become unavailable due to a number of reasons and the whole service provided by them is lost at that moment. Cloud has made this issue even more prominent. Large quantities of services depend on a handful of cloud providers which in turn rely on large centralized data centers.

Amazon's AWS outage in North America in 2017 is a textbook example of single point of failure: an issue in one data center rendered hundreds of services unusable [52]. This issue could be avoided by distributing the centralized services which depended on that data center across the world to different data centers and effectively making the services less centralized. However, considering the issues caused by the AWS outage, geographical distribution is not often implemented.

In contrast to client-server architecture, peer-to-peer networking does not rely on centralized servers [46, p. 73-74]. Peer-to-peer networks are decentralized distributed systems in which every client, later referenced as node, acts as both client and server. Node can request resources from other nodes and in turn serve other nodes. Peer-to-peer networks form a new overlay network layer on top of the existing physical network infrastructure.

Peer-to-peer networks avoid many drawbacks of the centralized client-server approach. There is no single point of failure which makes peer-to-peer networks resilient. Additionally, peer-to-peer networks are highly scalable: when the overlay network grows in size, the resource usage growth rate is less than linear [8, p. 7]. Full decentralization also eliminates the need for central authority or service provider and effectively removes all hosting infrastructure expenses.

2.2 Peer-to-Peer Overlay Networks

Peer-to-peer networks are based on logical overlay networks. These overlays are implemented on the application level and they add an additional layer of abstraction on top of the existing physical networks. Overlays are virtual: they do not require any new equipment to be installed and there is no need to modify existing software or protocols. The overlay network topology determines the type of the peer-to-peer system.

Peer-to-peer networks are often classified into unstructured and structured networks [8, p. 10]. The structure refers to the node and content placement in the network topology and on how the nodes communicate with each other. The peer-to-peer network properties such as performance and scalability depend on both the network topology and message routing algorithm.

In unstructured overlay networks a node connects to its neighbor nodes which it knows about [9, p. 50-54]. There is no predefined organized structure for node or data placement. Since a node knows about its neighbor nodes only it must communicate to the rest of the network trough the known nodes. Therefore, the structure of unstructured networks is often similar for example to random graphs or social networks.

The content search in the unstructured overlay networks is implemented via some routing algorithm. Basic routing algorithms in unstructured networks are flooding and random walk [9, p. 46-49]. Flooding works by sending a message, for example a search query, to all neighbor nodes. These nodes will then forward this message to their neighbors if they do not have the searched content available. This message has a time-to-live (TTL) value which is decreased in every node. The message propagation stops when this value reaches zero. Random walk works like flooding, but to decrease the message overhead, the message is sent only to one random neighbor.

Unstructured peer-to-peer networks can be divided into three different categories with varying amount of decentralization [46, p. 74]. These three network types are illustrated

in the Figure 2.2. Centralized unstructured networks have central servers which contain information about other nodes. The nodes use this information to connect to each other. Hybrid unstructured networks use dynamic central entities called supernodes to form a unstructured network. Normal nodes then connect to these supernodes. Fully decentralized or pure unstructured peer-to-peer networks consist purely of nodes which are all considered equal.

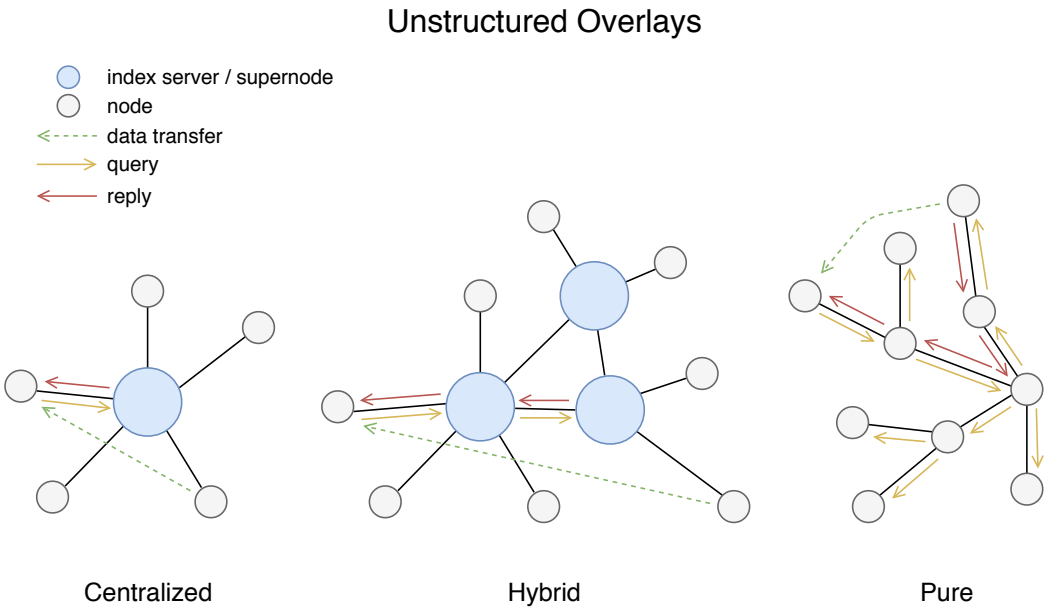

Figure 2.2. *Unstructured overlay network types.*

In contrast to unstructured networks, structured networks enforce some rules for the node and content placement in the network topology [9, p. 75-77]. Structured overlays can be categorized by the overlay network geometry and routing algorithm. The geometry defines the graph structure between nodes in the network. Certain geometry may support multiple different routing algorithms.

When a node joins to the structured network it will initialize a local routing table. This table is used by the routing algorithm to find content. Additionally, the node is given a virtual address from the virtual address space of the structured network. The address of node defines the place of the node in the overlay network geometry. The routing table and virtual structure enables deterministic routing and makes content search in the overlay network more efficient than in unstructured networks. However, the overlay network structure does not reflect the actual physical network structure. While this increases fault tolerance, the overlay network latency will increase if the nodes are distant in the physical network topology. [8, p. 13]

Most of the structured peer-to-peer networks use key-based routing algorithms [19, p. 226-227]. Distributed hash table (DHT) is often used to form structured overlays and to enable key-based routing. Each node in the network is responsible for a piece of the hash table according to some partitioning method. Every node is assigned an identifier (ID) which

defines the range of the table it is responsible for. When some data is added to the network, a hash is computed for example from the filename of the data. The hash along with the data is stored in the overlay network to a node which is responsible for storing the content. This node is determined by the calculated hash and node ID. A simplified illustration of a structured network can be seen in Figure 2.3.

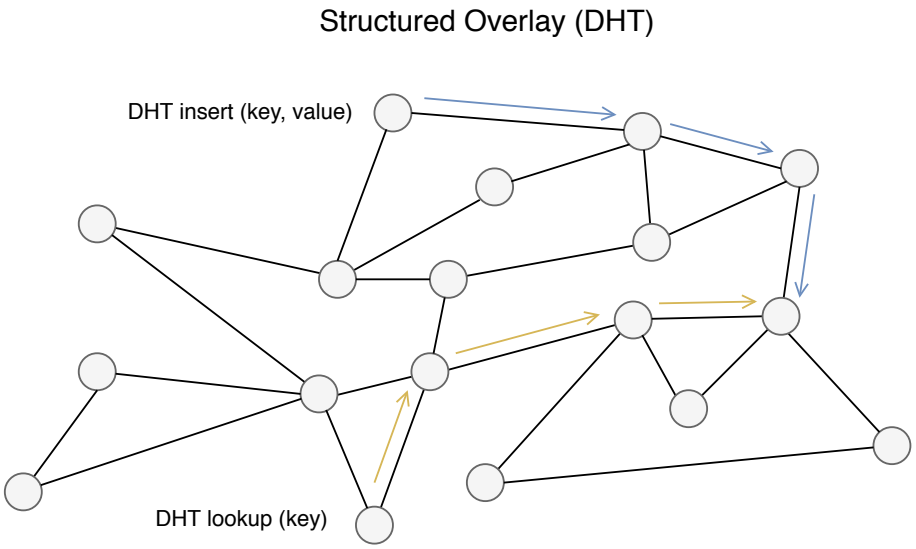

Figure 2.3. *Structured overlay network.*

After the content has been added to the structured network, it can be searched with the hash [19, p. 226-227]. The routing algorithm then finds the node which has the content by forwarding the query to a node which is closest to the target node's ID. DHTs drawback is that the routing tables must be maintained due to changes in the network [8, p. 19]. The process of constantly joining and leaving nodes is called as churn. Churn is common property for all peer-to-peer networks but in the case of DHT high churn rate means that the overlay network must constantly adjust the routing tables.

The four first DHT architectures which emerged from the research community were Chord [62], Pastry [58], CAN [55] and Tapestry [66]. Distributed hash tables have been since researched extensively and currently there exists tens of different DHTs. One of them is Kademlia [49] which is used in many modern peer-to-peer file sharing systems. More comprehensive information about distributed hash tables can be found in various textbooks, for example [9] and [65].

2.3 Peer-to-Peer File Sharing

Many of the original peer-to-peer network implementations were focused on file sharing. Many of them still exist and are being developed further. To understand the incentives behind modern peer-to-peer networks and decentralized solutions, some of the popular large scale file sharing peer-to-peer networks are introduced next in more depth.

One of the first large scale examples of peer-to-peer networks was file sharing system Napster [46, p. 75]. Napster utilized a centralized unstructured approach where nodes transferred files between each other but the file directory was stored in a central server. The centralized file directory contained metadata about the files and the nodes which had those files available. Nodes could then query the server for the files and initiate a file transfer directly with the node which had the file.

BitTorrent is another more advanced example of the unstructured yet partly centralized approach. There are central servers, trackers, which maintain node addresses which are sharing some file. To share a file, a special .torrent file must be created which contains the tracker address. These files can be published for example on a web site. BitTorrent client software then uses the .torrent file to initiate connection to the tracker. Using the information provided by the tracker server, client connects to the other nodes and starts the file transfer. The files are broken into smaller chunks in order to download the file from multiple nodes simultaneously. This enables also the possibility to start uploading the already downloaded chunks to other nodes before the whole file has been downloaded. [14]

The availability of files in BitTorrent is short-lived, since when nodes stop sharing certain file, it becomes unavailable. A study has shown that 38 % of the torrents become unavailable within the first month [41]. In addition, the BitTorrent protocol has so called free riding problem: users download files but don't share them. While the BitTorrent protocol has countermeasures against free riding, it is possible to download files without contributing anything back [47]. Despite these issues, BitTorrent is still one of the most popular file sharing peer-to-peer networks.

Like in any other centralized architecture, central servers introduce single point of failure to the system also in partially centralized peer-to-peer networks. Similarly, the central metadata or tracker servers cannot be scaled efficiently. The bandwidth overhead of the tracker server has been recognized in BitTorrent, but based on measurements, it is not considered as a bottleneck [14]. To address the issues of the partially centralized approach, fully decentralized peer-to-peer networks were developed.

One of the first fully decentralized peer-to-peer networks was Gnutella [46, p. 76]. To connect to other nodes in the Gnutella network a client application is needed. The client application tries to establish connection to a predefined amount of nodes upon first startup and assigns found nodes as neighbors. The Gnutella network makes it possible to search content by querying the network. This is similar to Napster, but in contrast to Napster's centralized metadata servers, the query is distributed through the Gnutella network with a flooding algorithm.

One of Gnutella's issues is that the content discovery method generates a lot of excess traffic by flooding search queries [56]. Despite being a fully decentralized network Gnutella's first versions couldn't scale due to the protocol's performance issues. This observation led later to redesign of the Gnutella protocol which is discussed later in this section. In addition to

the scalability issues, Gnutella suffers from free riding: about 70 % of the users are free riders [1]. Unlike BitTorrent, the Gnutella protocol does not have any countermeasures against free riding.

Gnutella was not the only fully decentralized peer-to-peer network which emerged in the early 2000's. Freenet is a decentralized file sharing system which has a special focus on encrypted data storage and strong anonymity [13]. One major difference to the Gnutella procotol is that all the nodes store not only their own content but also other content from the network. Due to the increased redundancy the network becomes more fault-tolerant.

Freenet does not support arbitrary file searches [13]. One-way secure hashes are used to uniquely identify different files. The file contents are encrypted: nodes are aware only of the file hashes but not the actual content of the files. To get a file from the Freenet network one must know the hash of the file. Freenet's routing algorithm resembles distributed hash tables but is less structured.

In Freenet each node maintains a routing table with known file hashes and node addresses. A node uses the table to forward the search query to a node which it assumes to be closest to the owner of the file determined by the lexicographical distance of the hash. The nodes are aware only of their immediate neighbor nodes. This makes the network more secure, because the nodes never know where the data is stored. The actual file transfer is not done directly with the node which has the file but through the Freenet network to protect the identity of the data owner node. [13]

While Gnutella was originally a fully decentralized network, the network structure was changed when Gnutella 0.6 was released [9, p. 55]. The redesign was required, because the flooding routing algorithm could not scale properly as described earlier in this section. The redesigned version was based on the hybrid unstructured overlay approach. The flooding algorithm remained the same but it was used only between supernodes.

Kazaa is another example of the hybrid unstructured network [46, p. 78]. The network structure is same as in Gnutella's latest version. Supernodes act as temporary index servers for the normal nodes. Kazaa client software is needed to join the network. The client offers a list of supernodes to connect to. After connecting to some supernode, the normal node sends a list of files it is sharing to the supernode. When a node requests some file, supernodes look first their local index for the file. If the file is not found, the supernode will query the file from the other supernodes.

Many of the distributed hash tables used in structured overlays were introduced only after the emergence of the previously described file sharing networks. Due to this the early peer-to-peer file sharing systems were based on the unstructured approach. Nevertheless, numerous modern peer-to-peer file sharing systems utilize DHT-based structured networks. Some of these file sharing systems are introduced and compared in the section 2.6.

2.4 Trust, Security and Privacy in Peer-to-Peer Networks

In the client-server architecture most of the security and data privacy concerns rely on the central authority. The central authority is trusted to keep for example private data safe and to otherwise protect the data which have been stored in the central servers. In peer-to-peer networks there is no central authority and thus every node must be able to trust the other nodes with which it is communicating.

Mutual trust among nodes is an important aspect of peer-to-peer systems. For example, free riding affects negatively to the trust in the network by distributing the load unevenly between the nodes. In addition to trusting each other, nodes must trust also that the actual shared content is authentic and has not been tampered with.

In content addressed networks such as Freenet there is a smaller change to receive wrong content due to the one-way hashes which identify the content cryptographically. If the content is identified by some other means, a malicious actor could easily spoof the content and distribute for example bogus data instead of the real file [9, p. 338]. Even in content addressed networks one must trust the original distributor of the content hash since it is impossible to know what the file behind the hash contains in reality.

Peer-to-peer systems have multiple attack surfaces on different functional layers of the system: application layer, overlay layer and network layer [9, p. 321-322]. Application layer attacks could be for example information leakage or excessive usage of the victim node's resources. Application layer depends on the overlay layer security. Especially in DTHs the routing tables are vulnerable to attacks and its important to properly secure them. Finally, the overlay layer depends on the actual network layer. This layer can be attacked by methods such as packet manipulation and interception.

Privacy is another important aspect of peer-to-peer networks and is closely related to both trust and security. The identity of the user is not usually protected in peer-to-peer networks in any way which enables for example user tracking [9, p. 337]. Some networks such as Freenet have been built so that they offer enhanced privacy and protection. Regardless, users must take care when adding content to the peer-to-peer network to avoid exposing sensitive information with the rest of the world.

Johnson *et al.* [40] argue that many issues related to security and privacy can be mitigated with good user interface design and user education. According to the authors, sensitive information gets commonly exposed for example due to a misplaced file, confusing user interface design or automated wizards. The user interface must be designed in a way which clearly shows what files are being shared. In the end, users must be educated to understand the risks of peer-to-peer file sharing and to use file sharing applications with care.

2.5 Peer-to-Peer in Mobile Environments

Due to the growth of the mobile device usage pervasive mobile environments are a common occurrence all over the world. In this thesis mobile environment refers to handheld devices which are connected to the Internet either via some broadband cellular network such as 4G or via wireless local area network (WLAN). The mobility of the devices introduces many properties which are not present on basic desktop computers. These properties have an effect on both to the peer-to-peer overlay network and handheld device performance.

The most restricting feature of mobile devices is that they are powered by a battery. The energy they can use is limited. If a device runs constantly a peer-to-peer network node the device will use more power which in turn affects directly to the uptime of the device. Additionally, the device may enter into a stand-by mode and disconnect from the overlay network at any given time [9, p. 300]. This causes additional churn to the peer-to-peer network.

Mobile devices connect to a different cell tower from time to time. The new tower may provide a new IP address to the device. According to Buford *et al.* [9, p. 299-300], peer-to-peer networks will usually interpret this as a leave-join sequence which will again cause more churn to the network. However, mobile devices have often the capability to be connected to the Internet via multiple interfaces at the same time. Connection redundancy might reduce the churn in some cases [9, p. 308].

Mobile device nodes are heterogeneous from the peer-to-peer network perspective: they have different capacities to store data and the performance might vary extensively amongst the nodes [9, p. 300]. The amount of data that can be transferred over the cellular network connection during a given time frame might be restricted. The speed of the network connection varies also significantly. All these variations have an effect to the overall performance of the overlay.

Hardware related properties are not the only limitations in mobile environments. The mobile operating system may impose some restrictions for the applications. A considerable amount of network churn might be caused if the peer-to-peer network node does not stay running in the background when the user proceeds to do some others tasks for example during a content download process. Therefore, the operating system's ability to run a fully capable peer-to-peer node in the background while using the mobile device for other tasks is essential.

Google's Android and Apple's iOS are the two major operating systems (OS) which dominate the markets currently. Apple has strict guidelines for applications which are running in the background [2]. Due to the restrictions it might be impossible to run a peer-to-peer network node in the background on iOS devices. Android has less restricted environment but running background applications is complicated since they can be forced to standby mode at any given time [31]. On the other hand, for example Sailfish OS [60]

offers more freedom for application development than the two dominating mobile operating systems. The restrictions of the mobile operating systems and the file sharing application implementation details are discussed more in Chapter 3.

2.6 Modern Peer-to-Peer File Sharing

Many new peer-to-peer content sharing networks have emerged in the last few years. Most prominent of these systems are InterPlanetary File System (IPFS) [6], Dat [54] and Swarm [63]. These three systems were selected to this comparison due to their popularity, open source implementations and the large development efforts behind them. Based on this comparison one of them will be selected to be used as the back end in the application implemented in this thesis.

InterPlanetary File System is a peer-to-peer distributed file system [6]. Its ultimate target is to replace Hypertext Transfer Procotol (HTTP) and build a new decentralized Internet infrastructure. IPFS combines multiple different proven techniques of past successful distributed systems to create a new solution which attempts to connect all computing devices under a single worldwide file system. Among these are for example Kademlia DHT, BitTorrent and distributed version control system Git [28]. The main idea of IPFS is to model all data in a single Merkle directed acyclic graph data (Merkle DAG) structure.

Dat is a peer-to-peer protocol which is designed for syncing versioned data [54]. It attempts to replace cloud based storage services such as Dropbox and Google Drive due to their centralized nature, high costs, slow transfer speeds and privacy issues. Dat has many similarities with IPFS: it is inspired by projects such as Git and BitTorrent. However, Dat's main idea is narrower: it offers a free and encrypted versioned peer-to-peer data syncing system.

Similarly to IPFS and Dat, Swarm is a distributed platform which provides peer-to-peer data storage services [63]. Swarm is closely related to the Ethereum blockchain ecosystem [25]. Its primary target is to serve as decentralized and redundant storage system for all data related to Ethereum. Unlike Dat and IPFS, Swarm does not require that the uploaders host the content themselves. This is possible due to a built-in incentive system which rewards nodes that are hosting content.

These three systems are compared from the perspective of the mobile application development. To be able to select the best system for the use case in this thesis and to be future proof, the following properties have been selected to the comparison:

- Initial release year
- Existing protocol implementations by programming language
- Developer community size, roughly estimated by the project's pages at Github
- Maturity of the software

Table 2.1. *Comparison of modern peer-to-peer content sharing systems. [37, 16, 26]*

Property	**IPFS**	**Dat**	**Swarm**
Initial Release	2014	2013	2016
Protocol Implementations	Go (reference implementation), JavaScript	JavaScript	Go
Community Size	Medium, 100+ contributors	Small, about 100 contributors	Large, hundreds of contributors
Implementation Maturity	Alpha, latest reference implementation version 0.4.17	Stable, latest version 13.11.4	Alpha (proof-of-concept), latest version 0.3

The selected properties have been collected to the Table 2.1 for each system. Clearly, both IPFS and Dat have been published a few years before Swarm. Due to this, IPFS and Dat projects have had more time to develop and test the system. Dat is the only project which has been able to achieve a stable release. However, despite IPFS being still in alpha state it is being used all around the world. The IPFS reference implementation, go-ipfs, is mostly functional but it lacks some features of the original specification [37]. On the other hand, Swarm is only at a proof-of-concept phase and is more likely to undergo larger changes in the future.

Swarm is backed both by a very large user and developer community because it is tightly related to the Ethereum blockchain ecosystem. IPFS has a medium-sized community behind it and is being actively developed. Dat has the smallest developer community, but the development is active and it is unlikely that the developers would abandon the project at this point.

The main implementations of IPFS and Swarm are written in Go. Go is an open source programming language [29]. It makes it easy to run both IPFS and Swarm almost on any device. In addition to Go, IPFS has also a JavaScript implementation which makes it possible to run IPFS in web browsers. The development efforts of Dat are focused solely on JavaScript. Like IPFS, Dat can be run on web browsers. Outside browsers Dat needs some JavaScript runtime or browser based environment to be able to run. This is a drawback, since it makes it harder to develop applications especially for mobile devices due to the special runtime environment needs.

From portability point of view Go is a lot better choice than JavaScript: compiled Go applications produce a single binary and if needed, applications written in Go can be compiled into standard shared libraries. This allows Go applications to be used easily as a library directly from other programming languages. It is also straightforward to cross-compile Go programs to other processor architectures which is often the case when targeting mobile environments.

While all the three projects are similar on higher level, their approaches to peer-to-peer

content sharing are very different when observed more closely. Swarm is part of the large Ethereum ecosystem but it means that it requires Ethereum to work properly. Additionally, Swarm is not as mature as the other two projects. Dat has had multiple stable releases, but JavaScript makes it hard to integrate it into resource constrained mobile environments. The remaining candidate is IPFS. Go-ipfs, the reference implementation of IPFS, can be considered stable enough for a proof-of-concept mobile application in this thesis. The next Section 2.7 will go trough the IPFS protocol and architecture in more detail.

2.7 InterPlanetary File System

InterPlanetary File System is a peer-to-peer network protocol which has been originally developed by Juan Benet [37]. IPFS integrates multiple different previously proven techniques together to create a distributed and versioned file system with no single point of failure. This section goes through the basics of the architectural stack of IPFS and explains how IPFS works.

The IPFS stack can be divided into five sections [37]. At the bottom there are three different layers: networking, routing and data exchange. These layers have been implemented in a separate library called libp2p [45]. The library provides modularized and extensible networking stack for peer-to-peer application development. The library was originally developed as a part of IPFS, but it was moved into external library to be able to leverage it also on other systems.

The first layer, networking, contains support for point-to-point transport between peers via protocols such as transmission control protocol / internet protocol (TCP/IP) and user datagram protocol (UDP). The next layer is the routing layer. The routing layer is responsible for enabling mechanisms to locate other nodes and content in the network. A solution based on Kademlia DHT has been selected in IPFS as the routing layer implementation [6]. However, the routing layer is implementation agnostic and the distributed hash table could be replaced with some other routing implementation.

The third and last layer included in libp2p is the data exchange layer. This layer is responsible for negotiating how the actual data transfer between nodes works. In IPFS the data block exchange protocol is BitSwap which is a new solution inspired by BitTorrent [6]. Like the routing layer, also this layer is implementation agnostic and could be replaced with some other data exchange protocol.

On top of the three bottom layers there are the data structure layer and the naming system layer. The data structure layer is implemented as a Merkle DAG [6]. Additionally, the files are versioned with a object model similar to the one used in Git. The underlying data model which describes these structures is implemented as a common hash-chain format called InterPlanetary Linked Data (IPLD) [39].

Merkle DAG is the main concept of IPFS: it creates links between the stored objects via cryptographic hashes. Merkle DAG makes the content in IPFS immutable and content

addressed. Due to this, content authenticity verification is simple. Additionally, objects with the same content are considered equal and stored only once.

The naming system layer, InterPlanetary Name Space (IPNS), enables mutable naming of the stored objects [6]. This is done by creating mutable pointers to the Merkle DAG. The naming scheme is inspired by Self-certifying File System (SFS) [50]. Every node is assigned a mutable namespace defined by the hashed node ID. Users can then publish an object under this namespace which points to the actual content in the Merkle DAG. The content is signed with the node's private key and thus the authenticity of the content can be checked with the public key and node ID.

Applications can be implemented on top of the IPFS protocol. Go-ipfs is the main reference implementation of the IPFS protocol [37]. Besides being able to run a full IPFS node, go-ipfs can used also as an application to control the node. For this purpose go-ipfs includes a command line client, an application programming interface and a web-based graphical user interface. An overview of the described architecture is illustrated in the Figure 2.4.

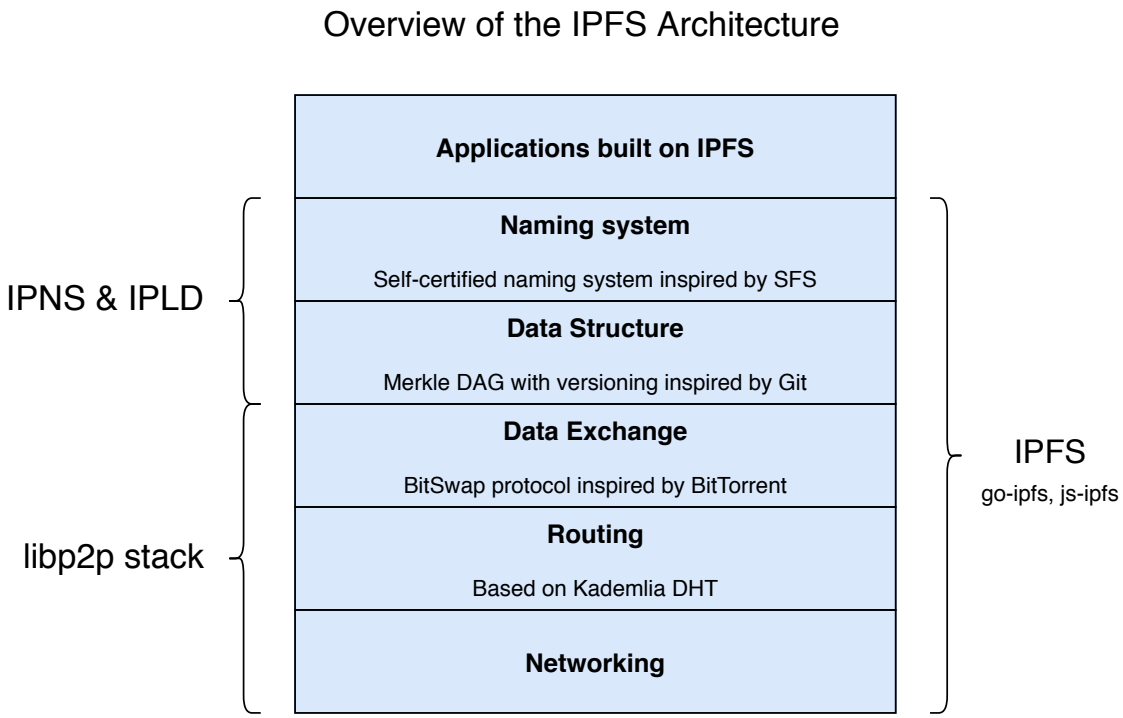

Figure 2.4. *Overview of the IPFS architectural stack. [37, 6]*

To understand how IPFS works in practice, some commands of the go-ipfs command line client application are introduced below. These commands give a good overview of the basic features of IPFS. There are multiple other more advanced commands available related for example to the IPFS data structure, naming system and network. More comprehensive overview of the features can be found from go-ipfs documentation [30].

Go-ipfs is available as a pre-built package for all common operating systems and architectures. After go-ipfs has been installed, the *ipfs* executable can be used as a command line

application. To start using IPFS, a local IPFS repository and configuration must be created. This can be done with init command:

```
$ ipfs init
$ generating 2048-bit RSA keypair...done
$ peer identity: QmcvdUhW2ZkHsbwAVsZd7czvbnPDyUBXF3HNtJeAMw6nTX
```

To add content to IPFS, *add* command should be used. This command works for both single files and folders. For this example, a dummy file with some dummy content is added to the IPFS.

```
$ echo 'This is some sample content' > ipfstest.txt
$ ipfs add ipfstest.txt
$ added QmU8r3op1FELD22uqUAVCyktkt1WStB3T4dExHEm6yPVJp ipfstest.txt
```

As seen above, the *add* command adds the file to the IPFS. To see the contents of the file, *cat* command can be used.

```
$ ipfs cat QmU8r3op1FELD22uqUAVCyktkt1WStB3T4dExHEm6yPVJp
$ This is some sample content
```

To download the contents of the file, *get* command can be used. The *get* command might be better than *cat* if the requested file contains binary data.

```
$ ipfs get -o testout.txt
   QmU8r3op1FELD22uqUAVCyktkt1WStB3T4dExHEm6yPVJp
$ Saving file(s) to testout.txt
$ 35 B / 35 B [=======================================] 100.00% 0s
$ cat testout.txt
$ This is some sample content
```

To list objects behind a certain IPFS hash, *ls* command can be used. For this purpose, a dummy sample folder with the a sample file with the same content as above will be created and added to IPFS.

```
$ ipfs add -r sample
$ added QmU8r3op1FELD22uqUAVCyktkt1WStB3T4dExHEm6yPVJp sample/sample.
   txt
$ added QmULnQ6ndnJJfLVnBABBRCqFKB4RXxMW88odPWJdPB2rxD sample
$ ipfs ls QmULnQ6ndnJJfLVnBABBRCqFKB4RXxMW88odPWJdPB2rxD
$ QmU8r3op1FELD22uqUAVCyktkt1WStB3T4dExHEm6yPVJp 35 sample.txt
```

In the example above the deduplication feature can be seen in action. The added folder contained the same file as we added before. The hashes are identical despite the fact the file names were different. However, from usability point of view it is very hard to remember any of these hashes. IPFS command line client offers an approach where single files can

be added to the system so that they are wrapped into a new folder and thus preserving the filename of the single file.

Wrapping the files into a meta data folder is still somewhat limited approach. Most of the users have accustomed to normal file system operations, namely moving files around folders, copying them to different locations, renaming them and deleting them. Due to the immutability of the data the described usage pattern requires a separate local virtual file system on top of IPFS. This virtual file system has been implemented in go-ipfs and it is called as mutable file system (MFS) [30].

MFS allows users to operate with the added files in a Unix-style manner by hiding the IPFS identifier based content addressing. The virtual file system references to the data in the IPFS. For example adding a file to a existing folder in the IPFS is a complicated operation because it changes the identifiers and the Merkle DAG of the file tree effectively creating new objects to the IPFS. MFS automates these operations and simplifies IPFS usage by providing file system like interface for the added contents.

The files added by the command line client add command are automatically pinned [30]. Pinning is an important concept of the IPFS: if a file has not been pinned, it will be garbage collected from the node repository at some point. Pinning ensures that the added content stays available at a node indefinitely. A node can select to pin any content from IPFS.

A node can share only content it has added or downloaded previously. If a node adds a file and does not pin it, the file will eventually disappear from the IPFS. However, the file will be cached to all other nodes which have downloaded the file. Unless some of the nodes pins the file, the cached file will be garbage collected at some point also from the other nodes.

In addition to local nodes, IPFS can be accessed via public gateways. These gateways are regular IPFS nodes but they provide a public HTTP interface to the IPFS. In practice, files in IPFS can be accessed directly from a web browser without any external application. For example, the file which was added in the previous examples could be downloaded directly via the official IPFS gateway from:

https://ipfs.io/ipfs/QmU8r3op1FELD22uqUAVCyktkt1WStB3T4dExHEm6yPVJp

The previous examples demonstrated the basic usage of IPFS command line client. However, the same commands can be used in different contexts. The mobile application implemented in this thesis will not be utilizing the command line client but interfaces directly with the go-ipfs library. This approach is described in the Chapter 4.

2.8 Related Work

The need for peer-to-peer content sharing mobile applications was first recognized in a user study conducted by Matuszewski *et al.* [48] in Finland. In the study the attitude towards mobile peer-to-peer content sharing applications was analyzed. The survey results

suggested that there is a need for such applications. However, the study revealed that free riding attitude is common even if the only cost of running a peer-to-peer application was higher battery usage. Most of the users were willing to run the peer-to-peer application only for a short periods of time given that they are familiar with the people they are sharing content with.

Mobile peer-to-peer usage in Finland was measured by Heikkinen *et al.* [35] in 2007. TCP/IP traffic was measured in the GSM/UMTS networks of three major operators. Different peer-to-peer applications were identified by transport protocol port numbers. As a result, direct peer-to-peer traffic was not observed but instead 9-18 % unidentified traffic was detected. This unidentified traffic was categorized to be possibly originating from peer-to-peer applications. In addition to direct network traffic analysis, a panel study was conducted. The study revealed that only one mobile peer-to-peer client application, Fring, had significant usage levels across all participants.

The BitTorrent protocol and different implementations based on it has been studied widely in mobile environments in many different contexts. These studies include the creation of a mobile BitTorrent application by Ekler *et al.* [24], power consumption study by Nurminen *et al.* [53], proxy-based approaches on content downloading (Kelényi *et al.*[43] and [42]), the analysis of BitTorrent protocol reliability by Bori *et al.* [7], hybrid BitTorrent based peer-to-peer content sharing solution by Ekler *et al.* [23] and mobile BitTorrent client usage and content lifecycle analysis (Ekler *et al.* [22] and Csorba *et al.* [15]).

Ekler *et al.* [24] implemented a mobile BitTorrent client application for low end Nokia Series 40 based devices. The results show that even simple mobile devices are capable of running a full peer-to-peer node utilizing existing peer-to-peer networks. According to the paper, the challenges introduced by the mobile environment were related to battery consumption and network bandwidth usage. Low battery consumption was seen important from usability point of view. High 3G network usage costs were an issue when the application was implemented.

Measurements done by Nurminen *et al.* [53] proved that peer-to-peer content sharing is feasible from power consumption point of view. In the research the power consumption of a mobile BitTorrent client was measured. The power consumption was on the same level as mobile phone voice calls. It must be noted that the measurements were done in 2008: mobile devices and their properties, most notably power usage due to more powerful hardware, has significantly changed since then.

Bori *et al.* [7] studied the reliability of BitTorrent protocol in mobile environment. The study was based on a mobile BitTorrent application called DrTorrent. The application had 5000 active users at the time of the evaluation of the statistics. Bori *et al.* conclude that BitTorrent is generally realiable, but the client applications must be capable of handling different anomalies such as failed TCP connections and corrupted data.

A hybrid BitTorrent based content sharing solution has been also studied: Ekler *et al.* [23]

proposed a new content sharing solution based on BitTorrent and central servers controlled by mobile operators. The solution supports both mobile and PC clients. The central server and and operator's central unit help to distribute the content more efficiently which in turn helps to reduce service provider costs.

Two different BitTorrent client implementations, SymTorrent and MobTorrent, were also addressed in the paper by Ekler *et al.* [23]. When the implementations were compared, SymTorrent was faster than MobTorrent on both 3G and WLAN networks. According to th authors this is due to the fact that MobTorrent was implemented in Java ME and SymTorrent had been written natively in C++. The Java implementation suffers from the overhead of the Java Virtual Machine and socket handling limitations of Java ME.

The content lifecycle and client usage in BitTorrent network has been studied by the data provided by MobTorrent mobile application by Csorba *et al.* [15] and Ekler *et al.* [22]. In the papers, 2 and 3 to 4 years of data collected via the MobTorrent client was analyzed. The data suggests that mobile BitTorrent has become more popular every year. The results show also that there were more unfinished downloads than completed downloads caused by multiple simultaneous downloads of the same content. Three largest categories by accessed torrents were videos, applications and audio.

In addition to BitTorrent, also other peer-to-peer protocols has been studied from energy consumption point of view in mobile environments. Gurun *et al.* [33] studied Chimera peer-to-peer protocol with an application on a embedded device. Most of the energy was consumed by the wireless interface when it was waiting for transmissions in idle state. As a energy conservation technique the wireless card was switched to a low power mode when no network communication was observed. This approach resulted in power savings. As a result, the study suggested that peer-to-peer protocols and applications can be used on low-power embedded devices.

The general performance of DHT-based peer-to-peer overlays in mobile environments was measured by Chowdhury *et al.* [11]. Five different DTHs were analyzed: Chord, Pastry, Kademlia, Broose and EpiChord. The measurements were done in an simulated environment. According to the results, the best choice for a mobile peer-to-peer overlay under heavy churn is Kademlia due to its good 97 % lookup success ratio while consuming 316 bytes/s of bandwidth. Additionally, EpiChord performs also well by achieving a 63 % success ratio while consuming only 70 bytes/s of bandwidth.

Cherbal *et al.* [10] presented an improved Chord-based structured peer-to-peer approach to minimize physical distance of the nodes in a mobile environment. The approach is based on two-tier structure where main Chord ring contains supernodes and secondary Chord ring contains cellular nodes. The authors provide a mathematical analysis of the approach which proves that the approach is more efficient in mobile environments due to reduced physical distance of the nodes.

Khan *et al.* [44] introduced a peer-to-peer network data store, MobiStore, which is targeted

for mobile environments. The authors argue that there is no good peer-to-peer solution for mobile world due to churn and resource limitations of mobile devices. To overcome these issues a new peer-to-peer network was developed. MobiStore's design attempts to mitigate the previous issues by structuring the peer-to-peer network into redundant clusters of peers and by updating the routing tables with a gossip-based protocol.

Simulations done by Khan *et al.* show that MobiStore achieves 12 - 48 % better lookup success rate than MR-Chord and Chord based peer-to-peer systems. Additionally, due to good churn and workload adaptation capabilities, MobiStore is capable of distributing requests more evenly than the two other systems.

In summary, the previous work has studied peer-to-peer networking in mobile environments from various different perspectives including topics such as peer-to-peer usage level measurements, mobile BitTorrent clients, power consumption measurements and different DTH-based systems. This thesis aims to extend the previous research by providing a fresh practical approach on peer-to-peer networking on mobile devices. Further, by creating an actual mobile application on top of an existing modern peer-to-peer network the behavior and performance of the system can be evaluated in real environment.

3. FILE SHARING MOBILE APPLICATION

This chapter focuses on defining the IPFS mobile client application implemented as a part of this thesis. First, the use cases for the application are discussed in Section 3.1. Based on the use cases a list of requirements is set in Section 3.2. The requirements are evaluated from the standpoint of this thesis and from the standpoint of end users. Section 3.4 compares different mobile operating systems from the point of view of this thesis. Section 3.5 gives a brief introduction to the selected operating system and further motivation for the selection. Lastly, section 3.6 discusses about the risks and challenges related to the implementation of the file sharing application.

3.1 Use Cases

The main use case for the application in the context of this thesis is to provide stable enough implementation to be able to run performance measurements for a device while the application is running. However, the use cases of the application should not be limited only to this thesis: to provide more value than pure academic research the application should be usable also by a common person. Most of the following use cases emerge from the standpoint of a common user but many of them are important also to be able to execute the performance measurements effortlessly.

3.1.1 Background Execution

Background execution of the application is important due to the nature of the DHT routing. Starting and stopping the application constantly causes churn and makes the usage of the application slow. The application and the IPFS node must stay running and connected while the device is used for other tasks unless it is explicitly closed. Background execution is very important also from the measurement point of view: for example battery consumption will be most likely measured with another program while the file sharing application is running.

3.1.2 Managing Node Settings

User must be able to see common settings of the IPFS node. User must be able to change the settings. This use case is also important regarding the measurements done in this thesis: for example the DHT routing mode must be easily changeable.

3.1.3 Sharing Files

File sharing is the main use case for a common user. A user must be able to add a file or a folder into the IPFS via the application. After adding the file, user must be able to see and copy the content identifier of the added content to be able to share the added content with other users.

3.1.4 Retrieving Files

Given an IPFS content identifier, user must be able to retrieve the data behind it through the application. The user must be able to insert a content identifier to the application and download the content addressed by that identifier from the network.

3.1.5 Managing Files

User must be able to see and browse added files like in a regular file system. User must be able perform operations on the files. These operation include for example removing, copying and listing content.

3.1.6 Viewing Node Information

User must be able to see common information about the IPFS node. This information includes node identifier, node status and the amount of disk space consumed by the node.

3.2 System Requirements

The previous use cases define a set of requirements for the application. The requirements are divided into six parts ordered by their importance from the standpoint of this thesis:

1. The application can start and stop a fully working IPFS node.
2. The application is able to run in the background while other applications are running.
3. The application can display basic information about the node and its status in the user interface.
4. The application user interface can be used to change basic settings of the node.
5. The application user interface allows adding content to IPFS and retrieving content from IPFS.
6. The application provides a file system style interface for managing the added files.

The battery level and network usage measurements do not require that all the set requirements are fulfilled. To be able to do the measurements it is required that only the first four requirements are met.

To be able to evaluate the generic usability of the application requirements five and six have to be implemented. They are also important to implement if the application should be usable by a common person.

3.3 Existing IPFS Mobile Applications

There exists an Android IPFS mobile application called IPFSDroid [38]. It is targeted primarily for developers. IPFSDroid ships with a full go-ipfs binary and runs go-ipfs as a separate daemon next to the application. For iOS or Sailfish OS there are no known IPFS applications.

The software architecture used in the IPFSDroid application has many drawbacks. Handling the IPFS daemon as a separate process on a mobile device is complicated as the main application becomes responsible for the daemon process. In addition to the IPFS itself, the daemon starts a separate HTTP server. The application uses the HTTP interface provided by this server to access the IPFS. This adds extra overhead to all operations since the files must be transferred over HTTP to the IPFS daemon. On a mobile device this approach is suboptimal and makes the application architecture complicated.

3.4 Platform Selection

As described in the Section 2.5 the operating system for which the application is implemented may limit the functionality of the application. This is true especially for the background execution requirement. More importantly, also the the flexibility of the platform must be considered from the application development point of view. In this comparison it is assumed that an interface to IPFS is provided via a shared library. Due to these reasons, three different mobile operating systems were selected as candidates: Android, iOS and Sailfish OS.

Java is the main programming language of Android applications [31]. Android allows also low level application development in C++. This makes it possible to use external shared libraries via the C++ code. However, a C++ application would need a bridge to the Java code to be able use the normal Android application programming interfaces. As explained in 2.5, Android allows background task execution but there is no guarantee that the application is allowed to stay on indefinitely.

On iOS applications can be developed in Objective-C which makes it possible to link a shared library directly with the application [2]. Running long or indefinite background tasks is not allowed in iOS. Consequently, application with a IPFS node would most likely lead to rejection from the application store.

Sailfish OS uses C++ as the main programming language [60]. Therefore, shared libraries are supported natively by the environment. Sailfish OS allows indefinite execution of background tasks: all started applications run in the background unless explicitly closed.

Both Android and iOS could be used to implement the IPFS application but the application development and distribution would likely require much more work than on Sailfish OS. Additionally, Sailfish OS has a special focus on openness and user privacy [60]. These aspects align better with the mission of IPFS than the other two systems which depend heavily on massive centralized organizations. The next Section 3.5 explains the motivation for selecting Sailfish OS as the implementation platform in more depth.

3.5 Sailfish OS

Sailfish OS is a mobile operating system developed by Jolla [60]. Version 1.0 of the operating system was released in 2013. Sailfish OS has been actively developed since: the latest version of the operating system is 2.2.1.18 released in September 2018.

Sailfish OS is based on Linux kernel and resembles many desktop GNU/Linux distributions [60]. Despite being a mobile operating system it can be run on a variety of devices including desktop systems. The Linux kernel of different Sailfish OS mobile devices is usually based on a device specific Android kernel. In practice, most of the mobile devices which run Sailfish OS have been originally Android devices. Sailfish OS has been ported on these devices by hardware adaptation work done by Jolla or community members. A separate glue layer between Android specific device drivers and Sailfish OS makes it possible the reuse the Android driver binary blobs in Sailfish OS.

On top of the Linux kernel and hardware adaptation layer Sailfish OS has Mer middleware [60]. The middleware includes for example Qt framework and various system libraries. Sailfish OS is capable of running Android applications through a separate proprietary Android runtime. On top of these stacks there is a proprietary Sailfish user interface layer. The native applications and user interfaces are developed with cross-platform framework Qt by combining C++ and Qt modeling language (QML).

For the purposes of this thesis Sailfish OS can be seen to be the best platform for the IPFS application implementation. The following aspects of the operating system and the application development environment were considered to be better on Sailfish OS than on Android and iOS:

- True multitasking support. Applications run on Sailfish OS in the background as they run on desktop operating systems. Running applications are killed only in the rare case of out-of-memory event.
- Applications are programmed with C++. A go-ipfs based shared library can be used directly from C++ without extra software layers.
- Native access to full GNU/Linux system and terminal without rooting or jailbreaking the device. This makes system metrics monitoring straightforward.
- Access to many command line applications which would be hard to come by in the other two operating systems.

3.6 Risks and Challenges

As it was explained in Section 3.3, there is only one pre-existing IPFS mobile application. The application developed in this thesis will be based on a completely different approach which requires direct integration with go-ipfs. This can cause unforeseen problems which may either block the development or slow down the development significantly.

Mobile devices have processors which are optimized for mobile usage. These processors have different architecture than desktop processors: many mobile processors are based on ARM architecture. When an application is developed for a mobile device the development is done usually on a separate desktop system and not on the device itself. Therefore, the host must use a technique called cross-compilation to be able to produce application binaries which are compatible with the target system.

The cross-compilation of the go-ipfs library into a shared library with a proper interface can be seen as a substantial challenge. It is possible to cross-compile Go programs into shared libraries with a Go tool called cgo [29]. The setup of a cgo cross-compilation environment is not straightforward. This is especially true for Sailfish OS which does not have any specific documentation for creating Go-based programs.

Go-ipfs is in alpha state but the library has an internal application programming interface called coreapi [30]. The interface exposes some of the go-ipfs core functionalities for direct integration. However, the implementation is not mature and lacks many functionalities. Some of the features needed by the file sharing application are not yet exposed through the coreapi. Due to this the implementation must use also other internal interfaces which require deeper knowledge about the go-ipfs codebase.

4. IMPLEMENTATION

This chapter goes through the implementation of the IPFS based mobile file sharing application and its architecture. The first Section 4.1 of this chapter will give a high level overview of the application architecture. Two different programs programs were developed as a part of this thesis: go-ipfs wrapper library and a mobile application which utilizes the library. Because the application depends on the library the development was done in two phases.

In the first phase the wrapper library was written for go-ipfs which is the reference implementation of the IPFS protocol. The wrapper was implemented in Go and cross-compiled into a shared library for two different processor architectures. The implementation details of the wrapper library are explained in Section 4.2.

In the second phase the wrapper library was utilized in a Sailfish OS mobile application written C++ and QML. The last Section 4.3 focuses on the implementation of the actual mobile file sharing application. Additionally, a few sample images of the application user interface are presented.

4.1 Overview of the Architecture

The file sharing application architecture was designed keeping in mind the fact that the application will be running on a mobile device with restricted resources. As it was explained in Section 3.3, running a separate go-ipfs based on daemon is suboptimal and impractical approach in mobile environments. Additionally, shipping multiple executables in an application package to the official Sailfish OS application store is not allowed.

Instead of interfacing with a separate daemon process it was decided that the application would interface directly with go-ipfs via a separate wrapper library. In addition to easier application packaging, the direct integration also removes any overhead caused by the HTTP server interface of the go-ipfs daemon. The resulting library is not Sailfish OS specific: if needed, the library can be used also in other operating systems in the future.

The high level overview of the architecture can be seen in Figure 4.1. The resulting IPFS client application was named as Asterism. The go-ipfs wrapper library is called as Lipipfs and it is embedded to the application package. Together, these two programs form the full application package.

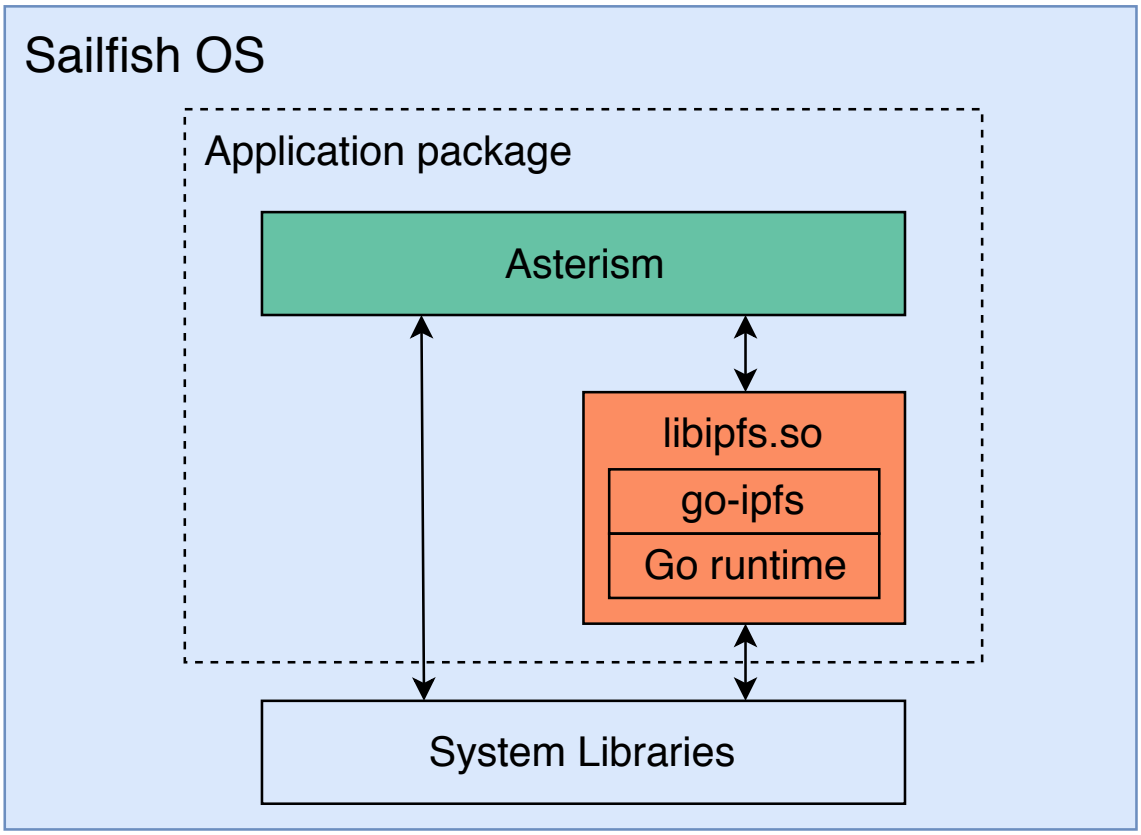

Figure 4.1. *Overview of the file sharing application architecture.*

Libipfs is dynamically linked to the main application which makes it possible to use Libipfs directly from C++ code. The Libipfs library contains full Go runtime and go-ipfs. The main application depends on some system libraries of which most are Qt framework libraries. Similarly, Libipfs depends on some C language specific system libraries.

4.2 Libipfs – Wrapper Library for go-ipfs

Libipfs, the IPFS library created for the file sharing application, provides a simplified interface to the underlying go-ipfs library. The Libipfs used in thesis is based on go-ipfs version 0.4.17 which was the latest version available during the development of the library. Go 1.11 was used to build the library. Libipfs is open source and can be found from https://github.com/skvark/libipfs.

Libipfs hides the complexity of the go-ipfs library behind a set of easy to use functions. The wrapper is written in Go, but the functions are exported for use by C code with the help of cgo. Cgo allows Go programs to interoperate with C programs. In Libipfs this means in practice that the Go functions are exposed to C programs. This makes it possible to call Go code from C applications.

Libipfs has also a single C function which is called by the Go code in certain cases. Many of the go-ipfs methods require interaction with the peer-to-peer network or may take otherwise long time to execute. Go has a built-in lightweight threading model called goroutines [29].

In Libipfs the exported functions which may take long time to run are wrapped in goroutines. Due to this, some of the functions called from the C code are by nature asynchronous and return immediately after calling them.

To handle the asynchronous functions Libipfs has a callback function written in C. This function is called from Go when a function wrapped in a goroutine has finished execution. The callback function calls a callback handler function originally passed as a pointer to the Go function from the application using Libipfs. The Go function results or possible errors as well as other information is passed back to the C or C++ code via the handler function. This makes it simple to run non-blocking functions from the application utilizing Libipfs as no separate threading functionality has to be implemented.

4.2.1 Cross-compilation for Sailfish OS Devices

To be able to use Libipfs on a mobile device it has to be cross-compiled to be compatible with the target devices. The Sailfish OS development environment provides cross-compilation toolchains for two different architectures: ARM and x86 [60]. ARM is more common architecture in mobile devices but x86 is required for example to be able to run the application in emulated environments.

Libipfs can be built with the help of the Sailfish OS software development kit (SDK). The Sailfish OS SDK can be installed locally to a variety of machines. It has a graphical user interface (GUI) for application development as well as a separate cross-compilation build environment and an emulator. The build environment and the emulator are provided as virtual machines. However, when automating the library build process the local installation of the SDK is not practical.

The Sailfish OS community provides separate container-based Sailfish OS platform SDK images [20]. The platform SDK is a separate environment which has a variety of development tools and virtual development environments for cross-compilation [60]. Libipfs builds are automated with the help of the Platform SDK container images.

Sailfish OS package management is based on the Red Hat Package Manager (RPM). All applications and libraries for Sailfish OS are distributed as .rpm packages. To be able to package Libipfs into .rpm file a separate libipfs.spec file was crafted. This file includes all information related to the package: setup process, build process and installation process among other things. The build command of the Sailfish OS Platform SDK uses the .spec file to build the library.

4.2.2 Continuous Integration

Libipfs is built automatically whenever there are changes in the code in the version control system. This practice is called as continous integration (CI). CI makes it possible to detect problems in the code earlier. The issues are also easier to spot because a failed build is

usually directly related to a small set of changes. Continuous integration can be combined with automated tests to further improve the quality of the software.

For Libipfs project the continuous integration pipeline is implemented with Travis [64] and Sailfish OS platform SDK container images. Travis is free CI service for open source projects and allows to use container images during the build process. Libipfs repository includes a .travis.yml file which configures the Travis build environment. The configuration file contains for example separate build matrix definitions for the ARM and x86 build targets.

The build section of the libipfs.spec file can be seen in Program 4.1. In short, the build process is following:

1. Git repository of the library is pulled into Travis CI service.
2. Several environment variables are set to configure the environment properly.
3. The Sailfish OS Platform SDK container image is pulled and started. Some parameters, such as Sailfish OS and go-ipfs version, are passed to the libipfs.spec file.
4. The build process is started as per instructions in the libipfs.spec file. The line numbers in the next sublist refer to the Program 4.1.
   (a) Go 1.11 binary is downloaded and installed (lines 1-7).
   (b) Go-ipfs and its dependencies are installed (lines 9-12).
   (c) A variety of environment variables are configured to be able to setup the cross-compilation environment correctly for Go (lines 16-36).
   (d) The library is built (line 38). The end result will be a shared library libipfs.so and the development header libipfs.h.
5. The resulting package is deployed to Github releases if a Git tag has been set.

The result of the build process can be downloaded from releases section of the Github repository. To use the library in the Sailfish OS SDK, the RPM package must be installed to the build engine. This can be done by logging in to the build engine via Secure Shell (SSH) and downloading and installing the desired RPM package:

```
1 curl -O -L https://github.com/skvark/libipfs/releases/download/0.1.0/
    libipfs-0.1.0-0.armv7hl.rpm
2 sb2 -t SailfishOS-2.2.0.29-armv7hl -m sdk-install -R rpm -i libipfs
    -0.1.0-0.armv7hl.rpm
```

After this the package is available in the Sailfish OS SDK given that the build targets have been refreshed via the Sailfish OS management tab in the GUI of the SDK. To use the package, the header file libipfs.h must be included to the referencing project.

```
1  cd $HOME
2  mkdir go1.11
3  curl -O https://storage.googleapis.com/golang/go1.11.linux-386.tar.gz
4  tar -xzf go1.11.linux-386.tar.gz --strip-components=1 -C go1.11
5
6  export GOROOT=$HOME/go1.11
7  export PATH=$PATH:$GOROOT/bin
8
9  go get -u -d github.com/ipfs/go-ipfs
10 cd $HOME/go/src/github.com/ipfs/go-ipfs
11 git checkout %{_go_ipfs_version}
12 make deps
13
14 cd $HOME/libipfs/src
15
16 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/srv/mer/toolings/SailfishOS-%{
       _sfos_version}/usr/lib
17 export CC=/srv/mer/toolings/SailfishOS-%{_sfos_version}/opt/cross/bin
       /%{_target}-meego-linux-%{_abi}-gcc
18 export CXX=/srv/mer/toolings/SailfishOS-%{_sfos_version}/opt/cross/bin
       /%{_target}-meego-linux-%{_abi}-g++
19 export GOOS=linux
20 export GOARCH=%{_goarch}
21
22 %if "%{_goarch}" == arm
23 export GOARM=7
24 %else
25 export GO386=sse2
26 %endif
27
28 export GOHOSTOS=linux
29 export GOHOSTARCH=386
30 export CGO_ENABLED=1
31 export CGO_CFLAGS_ALLOW=.*
32 export CGO_CXXFLAGS_ALLOW=.*
33 export CGO_LDFLAGS_ALLOW=.*
34 export CPATH=/srv/mer/targets/SailfishOS-%{_sfos_version}-%{_target}/
       usr/include
35 export LIBRARY_PATH=/srv/mer/targets/SailfishOS-%{_sfos_version}-%{
       _target}/usr/lib
36 export CGO_LDFLAGS=--sysroot=/srv/mer/targets/SailfishOS-%{
       _sfos_version}-%{_target}/
37
38 go build -x -v -ldflags=all="-s -w" -o libipfs.so -buildmode=c-shared
       go_ipfs_wrapper.go
```

Program 4.1. *The build section of the libipfs.spec file.*

4.3 Asterism – IPFS Mobile Client Application

Asterism is the IPFS mobile client application developed in this thesis. The application utilizes Libipfs presented in the previous Section 4.2 to provide a user interface for sharing files and interacting with the IPFS. Asterism was written for Sailfish OS in C++ and QML with the Qt framework. The features of the application are based on the system requirements set in the Section 3.2. Asterism is open source and available at https://github.com/skvark/Asterism.

The user interface of Asterism consists of four main views. The first view allows to view basic information of the IPFS node and makes it possible to start and stop the node. The second view provides an interface to the MFS with the possibility to add files to the IPFS. The added files are pinned automatically. The MFS can be browsed like a regular file system. Additionally, new directories can be created. Screenshots of these views can be seen in Figure 4.2.

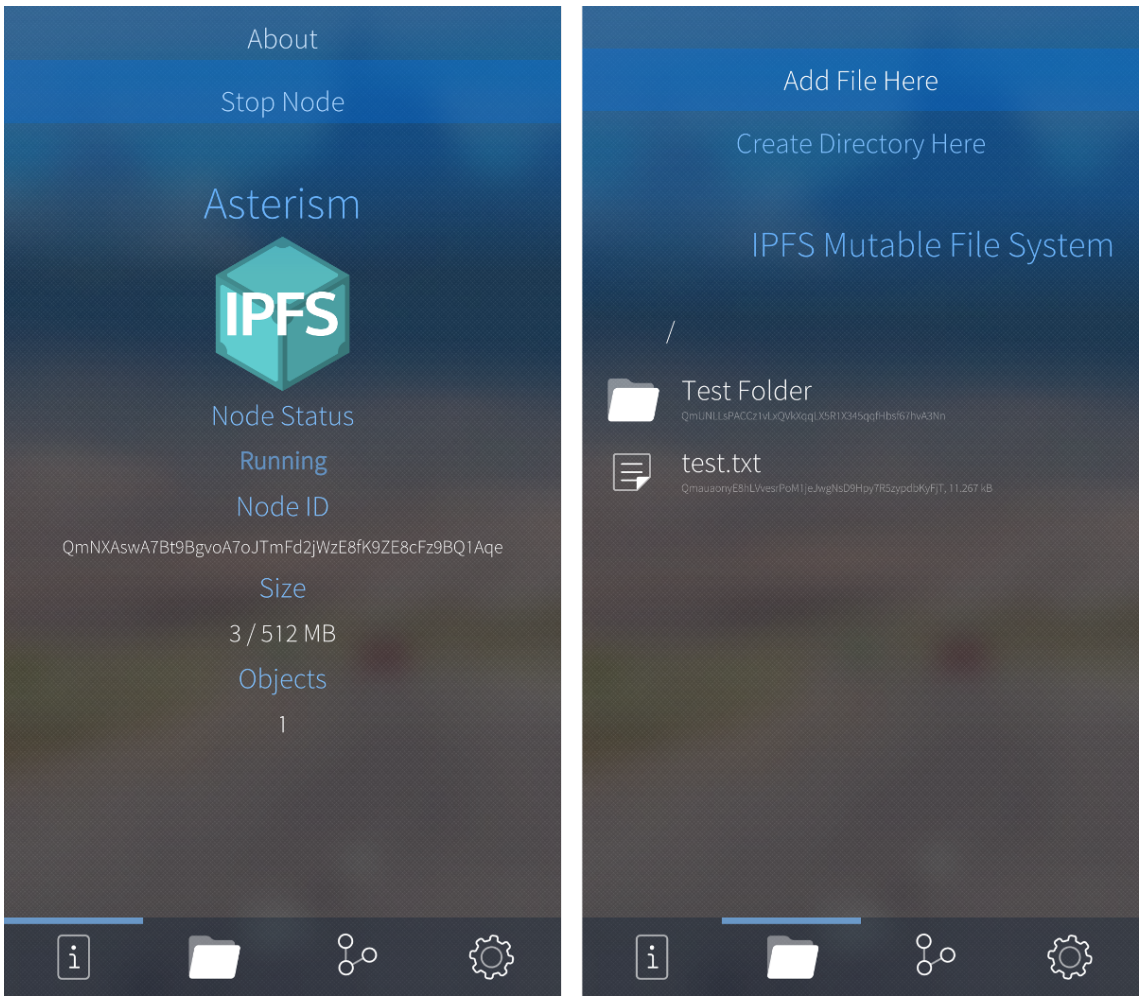

Figure 4.2. *Asterism info and MFS views.*

The third view includes a listing of the currently connected peers. The list includes the node identifier and address for each peer. Peer count could be limited via go-ipfs configuration file. Asterism does not provide user interface for changing the peer limit values.

The last view is the settings view. In the settings view user can change the mode of the IPFS node from full DHT mode to client only mode. This mode determines how the DHT works. In the client only mode the node does not serve requests to the rest of the network and should limit the network usage of the node to some extent.

In the settings view there are also settings for maximum repository size and manual garbage collection. The repository size limits the disk space used by the IPFS. Garbage collection allows to remove all unused or cached resources from the repository manually. An example of this kind of content are unpinned files. The garbage collections is also run periodically by the go-ipfs and when the maximum disk usage limit is approaching. Screenshots of the peers and settings views can be seen in Figure 4.3.

Special actions, such as stopping or starting the node, can be accessed via the pulley menu at the top of the screen. Asterism has a pulley menu in every view. The pulley menu is a part of the Sailfish OS user interface paradigm and can be accessed by pulling the screen down.

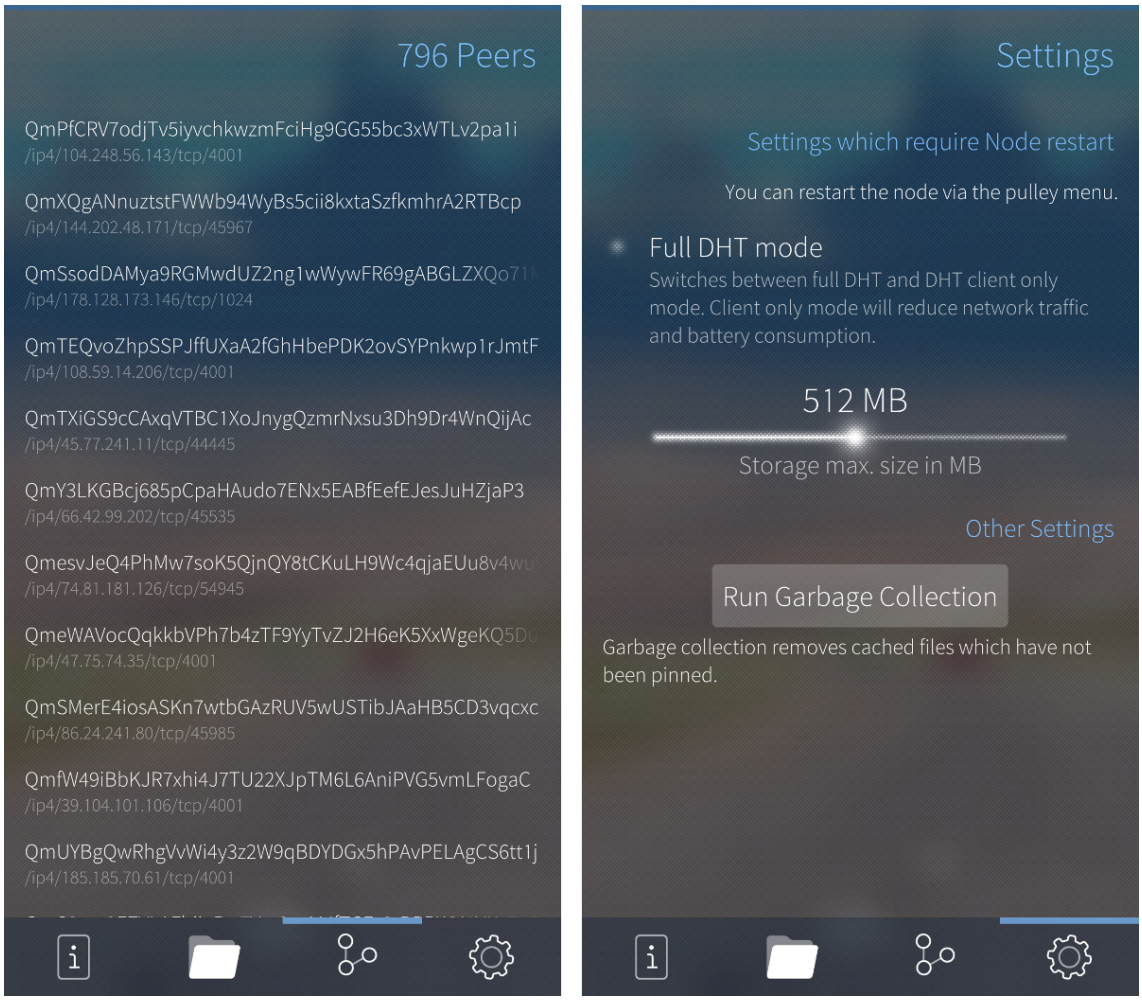

Figure 4.3. *Asterism peers and settings views.*

Asterism uses the exported functions of Libipfs to interact with the IPFS. An example of the file add sequence is visualized in the Figure 4.4. The add function and most of the other

exported functions are run in the background as goroutines. As it was explained earlier in this section, goroutines are lightweight threads of the Go programming language. These asynchronous operations are presented in the sequence diagram by the goroutine bounding box.

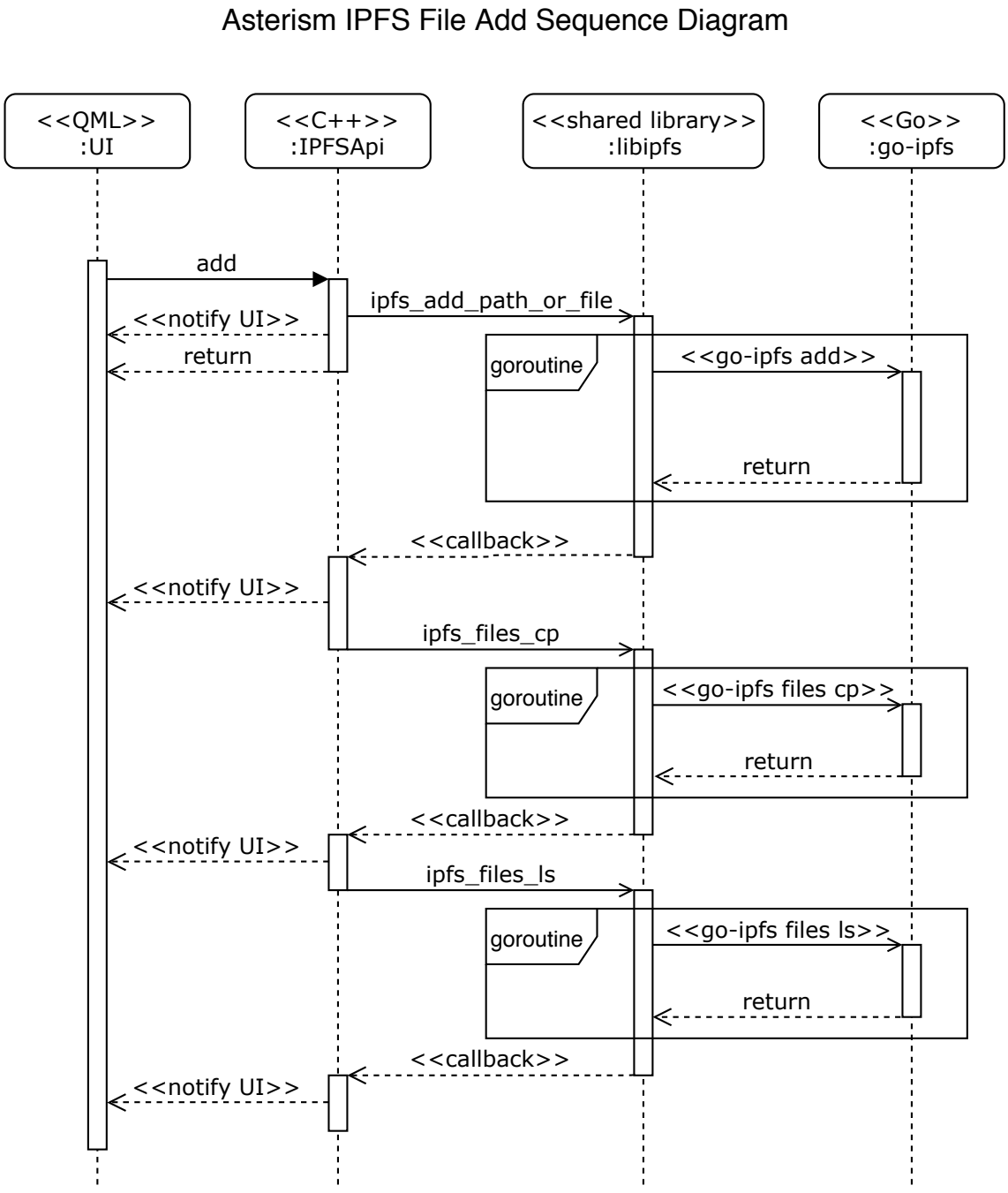

Figure 4.4. *File sharing application file add sequence diagram.*

To add a file to the IPFS, Asterism passes a file path to Libipfs add function which will in turn add the file to the IPFS with the help of go-ipfs. However, as it was explained in the Section 2.7, the return value of the file add operation is a hash identifier of the file contents. Effectively this means that the filename information has been lost during the operation.

Asterism uses the mutable file system implementation of go-ipfs to preserve the filenames and to provide a file system like interface to the added files. In practice, an entry by the original name of the file is created to the MFS by copying the hash which was returned from the add operation to the MFS. After the copy operation has been finished the MFS contents are refreshed by fetching the latest file listing from go-ipfs.

5. SYSTEM EVALUATION AND DISCUSSION

This chapter focuses on the evaluation of the file sharing application described in the previous Section 4. The evaluation in this chapter is based on the research question of the thesis: is it possible to transform mobile devices into decentralized distributed content delivery network without any central servers or authorities?

To answer to the research question, the application performance was measured on several different Sailfish OS based mobile devices. The performance measurements include battery consumption and network usage measurements. Additionally, the general functionality and usability of the application is evaluated.

The first Section 5.1 will introduce the devices and the performance measurement results. After the measurements in the Section 5.2 the application functionality is evaluated. Further, the results presented in the two previous sections are analyzed and discussed in the last Section 5.3.

5.1 Measurements

As it was explained in Section 2.5, mobile devices have many features which are more limited than on normal desktop computers. Peer-to-peer applications have a continuous need for data exchange caused by the data lookups and changes in the network routing. This will have an effect on the network usage: a peer-to-peer application will use network more actively than a normal centralized network architecture based application.

Consequently, active network usage will most likely have a negative effect on the battery life of a device. Therefore, both the battery consumption and network usage were measured while the file sharing application was running on a mobile device. With these measurements it can be evaluated is it worthwhile to run a peer-to-peer application on a mobile device.

The measurements were done on official Sailfish OS version 2.2.1.18. It was the latest available version when the measurements were performed. The go-ipfs version used by the Asterism file sharing application was 0.14.7. During measurements the Android support layer of the Sailfish OS was turned off to avoid possible measurement bias caused by the additional power consumption of the layer.

The battery consumption measurements were done on three different devices. The devices are listed in the Table 5.1 and pictured in Figure 5.1. These devices fall into different categories considering their price point and features. All three devices have been released during a timespan of one year in 2015 and 2016. Two of the devices are mobile phones and one of them is a tablet.

Table 5.1. *The properties of the devices used in measurements. [32, 18]*

Property	Sony Xperia X	Jolla C	Jolla Tablet
Release Date	2016, February	2016, May	2015, July
Chipset	Qualcomm MSM8956 Snapdragon 650	Qualcomm MSM8909v2 Snapdragon 212	Intel Atom Z3735F
Central Processing Unit (CPU)	4 x 1,4 GHz Cortex-A53 & 2 x 1,8 GHz Cortex-A72	4 x 1,3 GHz Cortex-A7	4 x 1,8 GHz
Battery	2620 mAh	2500 mAh	4450 mAh
Network Interfaces	GSM / HSPA / LTE & WLAN (Wi-Fi 802.11 a/b/g/n/ac, dual-band)	GSM / HSPA / LTE & WLAN (Wi-Fi 802.11 b/g/n)	WLAN (Wi-Fi 802.11 a/b/g/n, dual-band)
Talk Time	19 h	20 h	N/A

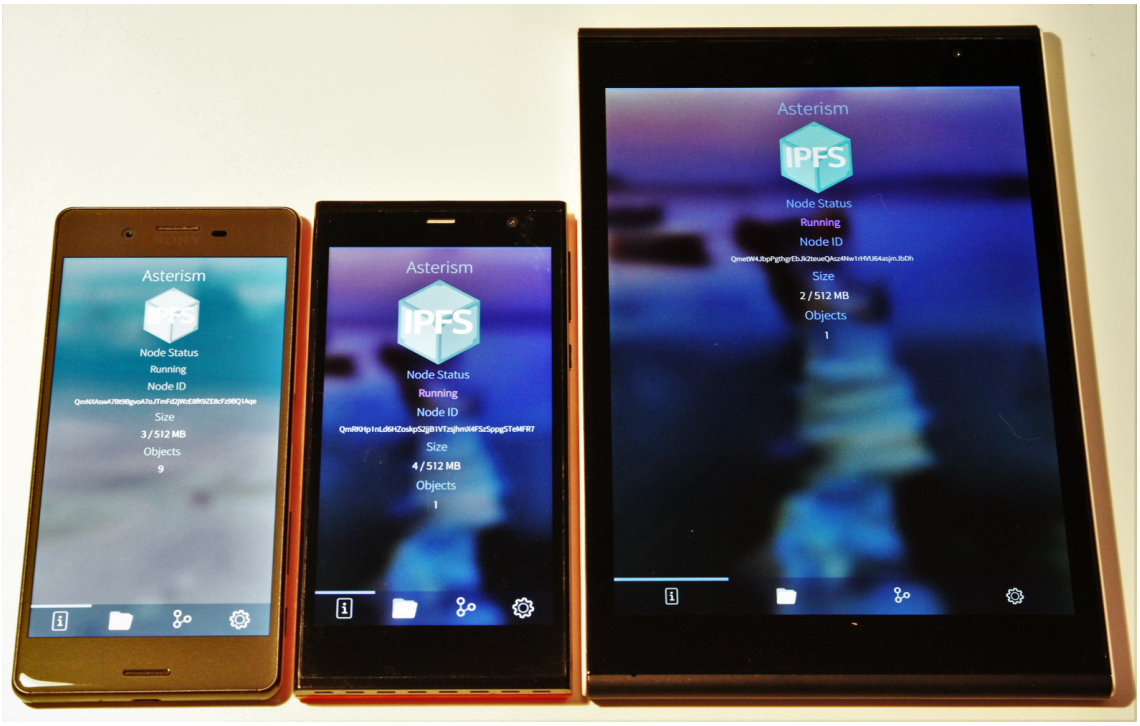

Figure 5.1. *The devices used in measurements: Xperia X, Jolla C and Jolla Tablet.*

The first device, Sony Xperia X, is the latest and most powerful of the devices. It can be seen as a mid-range device which represents most of the commonly used mobile phones. The second device is the Jolla Tablet. The tablet was selected for measurement to get samples also from a device which is not a mobile phone. On the tablet measurements were done only on WLAN interface because the tablet has no cellular connectivity. The last device, Jolla C, is a low-end mobile phone with a less powerful processor than the two other devices. Jolla C is also known as rebranded Intex Aqua Fish device.

5.1.1 Power Consumption

The power consumption measurements were executed in multiple different configurations. IPFS node was run in full DHT mode or in client only mode. The differences of the modes were explained in Section 4.3. The IPFS modes were combined with different network interface combinations: both 4G and WLAN enabled, only WLAN enabled and only 4G enabled. Additionally, the idle power consumption was measured for all devices while all network interfaces were enabled and no applications were running. During all measurements the screen of the device was turned off. No content was added to the IPFS and no content was downloaded from the IPFS before or during the measurements.

The power consumption was measured with Charge Monitor application available in the official Sailfish OS applications store. The application is able to log the battery level of the device into a log file. All measurements were started from a battery level of 100 %. Battery levels were logged for 4 hours. A linear regression line was fitted to the obtained data points. With the fitted line the battery life of the device can be estimated with a reasonable accuracy. The estimated battery life numbers in the next paragraphs have been rounded to 10 minute accuracy and the percentage differences in 5 % accuracy.

The firsts two figures, 5.2 and 5.3, include measurements which were done on the mobile phones when all network interfaces were enabled. The idle consumption of the devices is also included in these two figures. The battery of the Xperia X is estimated to last about 780 minutes in full DHT mode and about 850 minutes in client only mode. For Jolla C the same numbers are 620 minutes and 710 minutes.

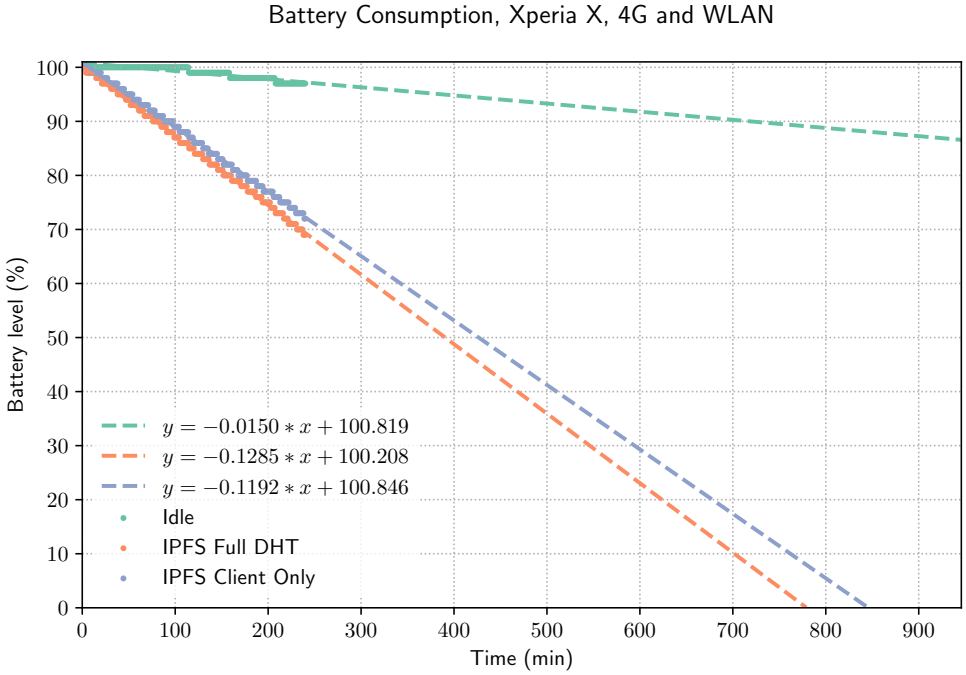

Figure 5.2. *Xperia X WLAN and 4G battery consumption.*

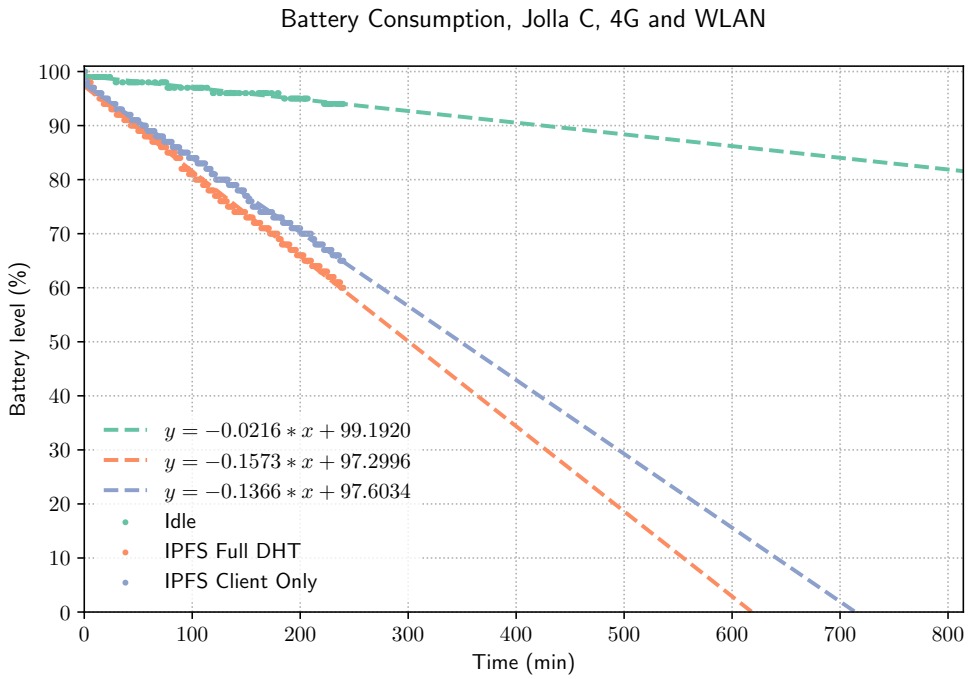

Figure 5.3. *Jolla C WLAN and 4G battery consumption.*

Both devices have similar power consumption trends. The idle power consumption is low which is also expected because no applications were running. The difference between the battery life of the devices is large. On full DHT mode the battery of the Xperia X lasts significantly longer than Jolla C's. Similarly, Xperia X performance is better than Jolla C's when the IPFS node is run in the client only mode.

The difference in power consumption between the DHT modes on the both devices is noticeable. On Xperia X the client only mode adds about 10 % more battery life when compared to the full DHT mode. On Jolla C the change is a bit larger, about 15 %. As the estimated battery lives are better than in the 4G only figures and worse than in WLAN only figures, IPFS node has probably defaulted to the WLAN interface during the measurements.

The next three figures, 5.4, 5.5 and 5.6, are WLAN only measurements. For Xperia X, the estimated battery life in full DHT mode is 880 minutes and in client only mode 1080 minutes. For Jolla C the estimations are 720 minutes in full DHT mode and 1020 in client only mode. For the Jolla Tablet the numbers are 720 minutes and 820 minutes.

The battery lives of the Xperia X and Jolla C are longer on WLAN only when compared to the measurements where both WLAN and 4G were enabled. The percentage changes between the DHT modes are about 20 % on Xperia X, 40 % on Jolla C and 15 % on Jolla Tablet. The Tablet results are similar with the two mobile phones. However, the results indicate that the WLAN interface of the Tablet is less energy efficient because it should last longer given the capacity of the battery.

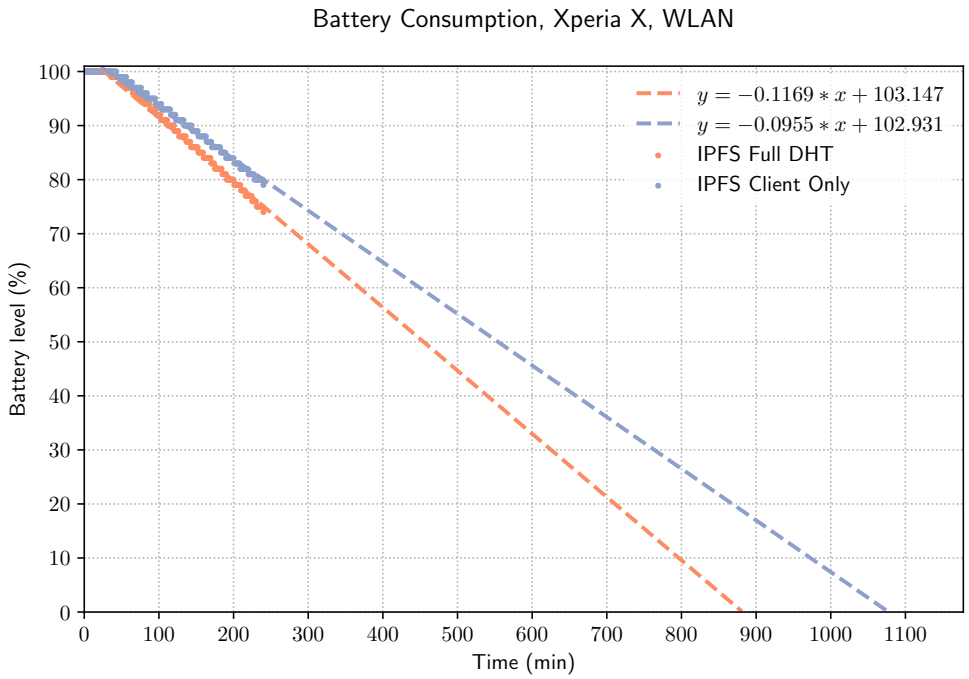

Figure 5.4. *Xperia X WLAN battery consumption.*

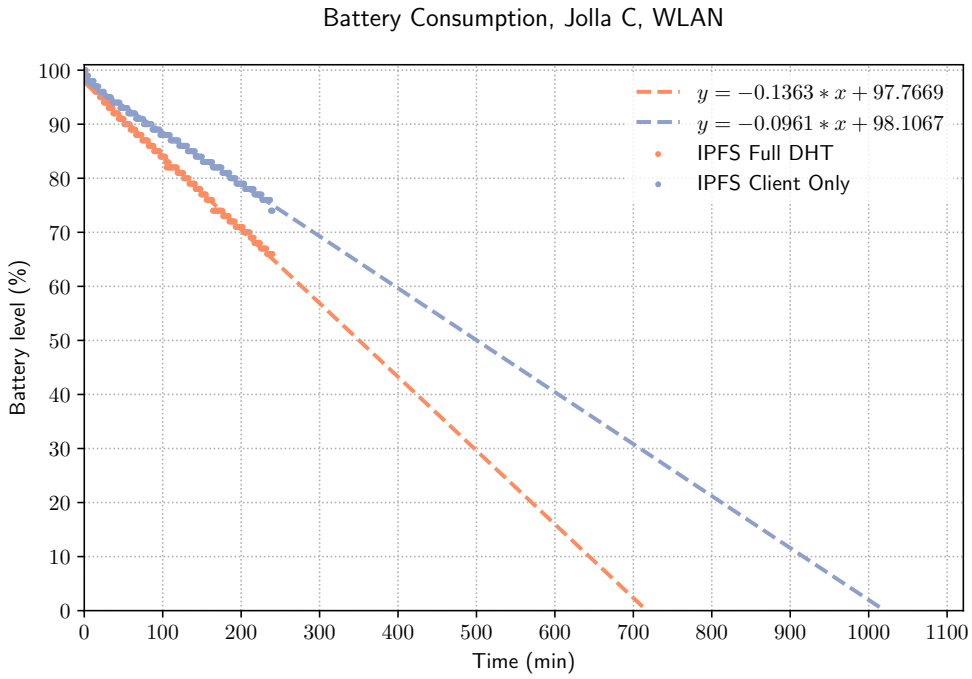

Figure 5.5. *Jolla C WLAN battery consumption.*

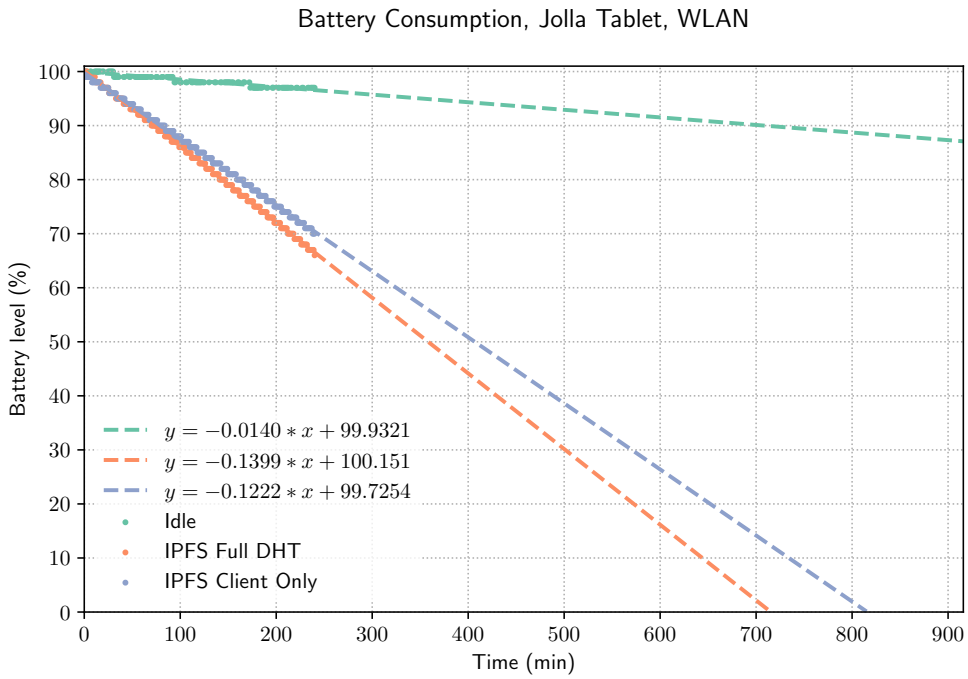

Figure 5.6. *Jolla Tablet WLAN battery consumption.*

The last set of power consumption measurements were done on 4G interface only. The results of these measurements are in figures 5.7 and 5.8. For the 4G only measurements, the estimated battery life for Xperia X is 420 minutes in full DHT mode and 430 minutes in client only mode. For Jolla C the same numbers are 420 minutes and 440 minutes.

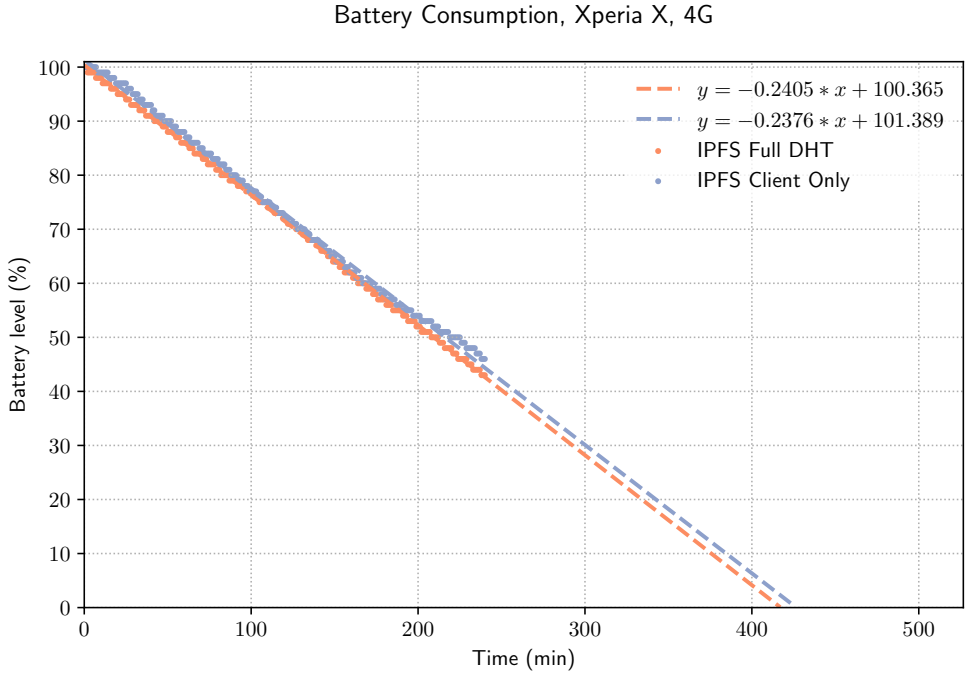

Figure 5.7. *Xperia X 4G battery consumption.*

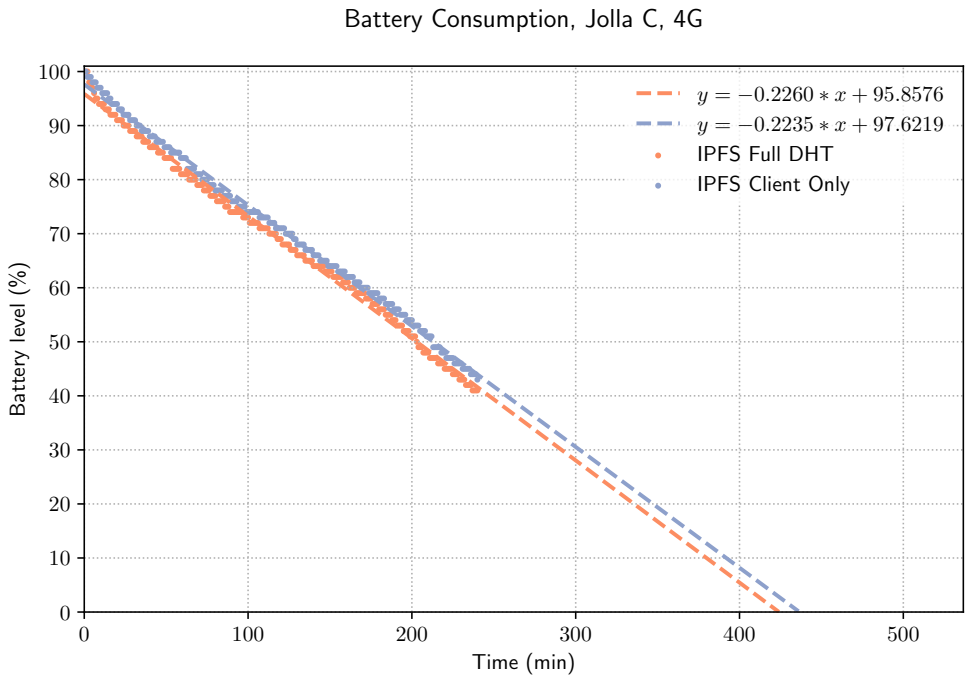

Figure 5.8. *Jolla C 4G battery consumption.*

The 4G only results have close to none difference between the different DHT modes. Running IPFS node on 4G clearly consumes much more power than when using only WLAN. Both devices perform very similarly on these measurements: battery life on both devices is approximately the same. The WLAN interface is clearly far more energy efficient than 4G on both devices.

5.1.2 Network Usage

The network usage measurements were done only on Xperia X. Network usage was measured in full DHT mode and client only mode with Nethogs [51] traffic monitoring tool. Both networking interfaces, WLAN and 4G, were turned on because it is the most likely default setup during common mobile phone use. After Asterism had been started Nethogs was run for 140 minutes. The total sent and received data was logged to a log file in MB every 10 seconds with the following command:

```
1   $ nethogs -t -v 3 -d 10 | awk '/harbour-asterism/{ print strftime("%Y
      -%m-%d␣%H:%M:%S"), $2, $3; fflush(stdout) }' > asterism.log
```

The results of the measurements are presented in Figure 5.9 and 5.10. Asterism downloaded in both modes approximately the same amount of data. Client only mode reduced the sent data about 30 %. The amount of the downloaded and uploaded data is very large. The application downloaded close to 800 MB in 140 minutes in both modes. Upload numbers were about 440 MB full DHT mode and about 300 MB in client only mode.

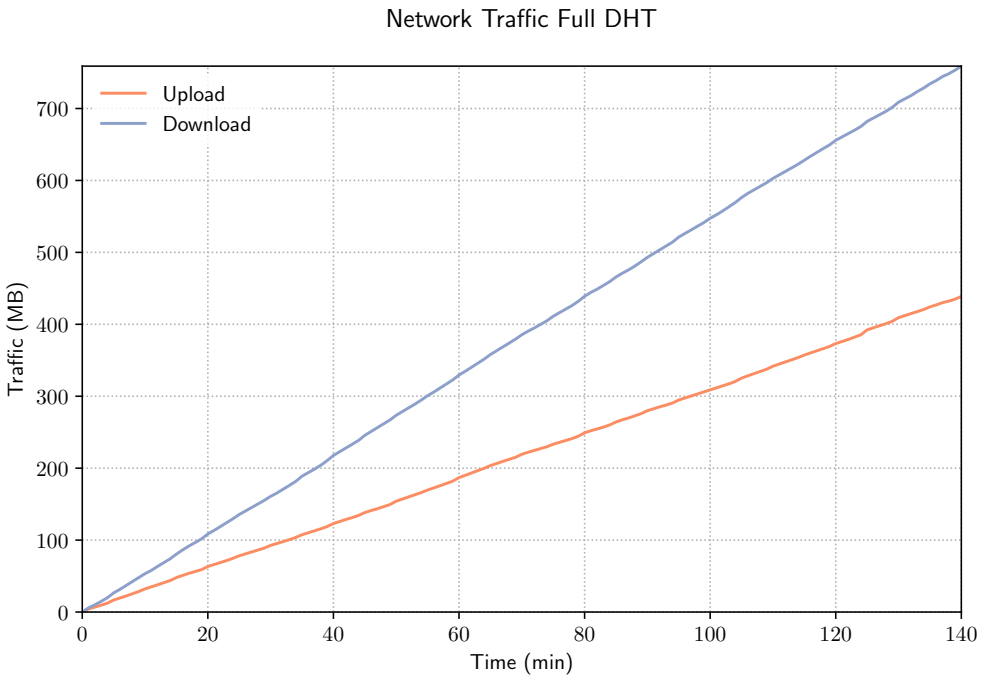

Figure 5.9. *Network traffic in full DHT mode.*

Figure 5.10. *Network traffic in client only mode.*

The networking layer used in go-ipfs was able to bypass Network Access Translation (NAT) and possible firewalls of the cellular network operators. This was confirmed by adding previously unknown content to the IPFS while running only on 4G. After this the content could be downloaded from a public IPFS gateway indicating that go-ipfs can upload content through cellular networks to the other nodes of the peer-to-peer network.

5.2 Functionality

The features of the application are based on the requirements set in the Section 3.2. The next listing will briefly evaluate if the set requirements were met.

IPFS node can be started and stopped. Asterism is able to start and stop the IPFS node. This can be done manually from the user interface (UI) at any given time.

Background execution is possible. As Sailfish OS is a true multitasking operating system, this requirement is met by default. User can leave the application running in the background or choose to close the application at any given time.

Node information and status in the UI. Node related information, such as is it running and the size of the repository, is displayed in the application UI. Go-ipfs includes also bandwidth monitoring functionalities, but they were not wrapped in Libipfs. Therefore Asterism is not able to show bandwidth statistics in the user interface.

Ability to change the basic settings of the node. The mode of the DHT routing can be changed as well as the maximum size of the repository. Additionally, user can run garbage collection manually. Peer connection limits as well as many other settings were not exposed via Libipfs and thereby not implemented in Asterism UI.

Content sharing and retrieval. Content can be added to the IPFS via the Asterism UI. Content download was not implemented as the download functionality was not needed to perform the measurements. However, download support is important to be able to use application truly as a file sharing application.

File system interface to the added files. Asterism implements a partial file system interface for the added files. New folder can be created and the virtual file system can be browsed. Deleting files and moving or copying files to another location was not implemented in the scope of this thesis.

The ability to run the IPFS node in the background and the ability to change the mode of the DHT were the most important functionalities in the scope of this thesis. Native background execution support made it possible to execute long running measurements. Access to the DHT routing mode in the UI enabled faster measurement cycles. The settings UI also decreased the possibility of configuration errors during measurements because manual node configuration via go-ipfs configuration file was not needed.

The implementation proves that a fully working IPFS node can be implemented and used on a mobile device by directly interfacing with go-ipfs. The continuous integration pipeline of Libipfs makes it easy to add more features to both Libipfs and Asterism in the future. More importantly, Libipfs enables the possibility to build IPFS based applications also on other resource constrained systems where the separate daemon approach is not practical or implementing Go-based graphical user interface is not a viable solution.

5.3 Discussion

The analysis and discussion in this section is based on the results in Section 5.1. The results indicate that running a full IPFS peer-to-peer network node on mobile devices is possible. However, the node causes large amounts of network traffic and affects negatively to the battery life of the device.

It is important to notice that no content was downloaded or uploaded during the measurements. All network traffic was caused by the chattiness of the IPFS protocol. DHT routing updates cause some of this traffic and it is likely that the IPFS Bitswap protocol is also responsible partly for the traffic. The amount of network traffic increases if a node downloads or adds some popular content and thus participates in the distribution of that content. This behavior is caused by the data exchange protocol which is similar to BitTorrent.

The go-ipfs implementation is not mature and there is a lot of room for optimization. On a mobile device the amount of traffic caused by the IPFS node is not acceptable in most cases due to the monthly upper traffic limits on mobile network subscriptions. Setting the DHT mode to client only has only minimal effect to the amount of the sent data. Therefore, traffic limit setting should be built to the IPFS protocol implementations to avoid excessive bandwidth usage.

Due to the constant network traffic the device can never enter to deep sleep mode. In the idle measurements the deep sleep behavior could be observed. The battery level logs contained longer gaps because the values were logged only when the device periodically briefly woke up to perform some tasks. The constant network traffic caused by the IPFS node prevents the device from entering to the deep sleep mode. Consequently, this behavior also prevents possible constant churn caused by the disconnections despite affecting negatively to the battery life.

The measurements were done on a stable environment. The devices were located close to a cellular tower and a WLAN router. Device movement and connection quality would most likely have an additional negative effect to the battery life. Additionally, mobile devices are usually used also for other tasks. Browsing the web or playing games while the IPFS node is running would consume the battery even faster.

The mobile BitTorrent measurements done by Nurminen *et al.* [53] had similar results when compared to the results of this thesis. Client only mode reduced the power consumption and the cellular network interface was less energy efficient than WLAN. The power consumption was on the same level with phone calls when run on 3G. In contrast, Asterism consumed more power than phone calls: the results indicate that on 4G the battery life is about one third of the talk time of the Xperia X and Jolla C. However, on 3G connection Nurminen *et al.* performed the measurements only for content download since the BitTorrent implementation could not bypass NAT and firewalls of cellular operators. Due to this, the results are not directly comparable.

Asterism was able to upload previously unknown content through the cellular network connection when it was queried through a public IPFS gateway. This is an important finding: if the mobile node is not able to upload content on cellular networks, the node cannot act as a full peer and only consumes the resources of the rest of the network. While the client only mode can be desirable due to the bandwidth and battery life savings the negative effect must be also considered. If a large amount of mobile client only IPFS nodes were to suddenly to join the IPFS, the network would not likely perform as well as before.

6. CONCLUSIONS

This thesis examined the possibility to transform mobile devices into decentralized content delivery network without any central servers or authorities. This was done by implementing a peer-to-peer mobile file sharing application and measuring how the application affects to the performance of the mobile device. InterPlanetary File System was selected as the peer-to-peer network and the application was implemented for Sailfish operating system.

The file sharing application was developed in two phases. In the first phase a wrapper library was written for the reference implementation of the InterPlanetary File System. With this approach a separate daemon process was avoided. A continuous integration and delivery pipeline was set up for the wrapper and it was cross-compiled for the target systems as a shared library.

In the second phase the shared library was utilized in the implementation of the mobile file sharing application. The application provides an easy to use user interface to the peer-to-peer file system. The user interface makes it possible to manage the added content in a way similar to common file system browsers. Additionally, the application is able to show basic information about the node and peer nodes as well as to control some settings of the node.

The network usage and the power consumption of the application was measured on three different devices. Two of the devices were mobile phones and one of them was a tablet. No content was added or fetched from the network during the measurements. All devices were able to run the application without problems. The application was able to upload content not only via WLAN but also through cellular connection.

The results show that the application consumed large amounts of network bandwidth. The traffic was caused by the chattiness of the peer-to-peer network protocol. Distributed hash table routing updates and the BitSwap protocol of the InterPlanetary File System are the most likely causes for the excessive bandwidth usage. When the distributed hash table routing mode was changed to client only mode the amount of the sent data was reduced by about 30 %.

Due to the constant network traffic the devices could never enter to deep sleep mode. This lead to large power consumption. On WLAN interface the battery of the measured devices was estimated to last about 720 to 880 minutes depending on device. When the distributed hash table routing mode was changed to client only mode the devices gained 15 to 40 % more battery life. When run on 4G interface only the estimated battery life dropped down to 420 minutes on both mobile phones which is about one third of the estimated talk time

of the phones. Client only routing mode had negligible effect on the battery life when the application was run only on 4G connection.

The amount of bandwidth consumed by the application is too excessive for continuous usage due to the data limits in usual mobile network data subscriptions. If a user has an unlimited data subscription the application could be run constantly in the background given that the user accepts greatly degraded battery life.

From usability point of view the application proves that mobile phones can be used as a part of worldwide peer-to-peer file system. Therefore, decentralized solutions could be potentially used in the future to replace centralized clouds and servers on mobile devices. However, for mainstream use cases the InterPlanetary File System needs network usage optimizations or bandwidth limits. Additionally, further research on the topic is needed when the protocol implementations have matured into stable releases. The results of this thesis have been summarized into a scientific article and submitted to the Seventh IEEE International Conference on Mobile Cloud Computing [36].

REFERENCES

[1] E. Adar, B. Huberman, Free riding on Gnutella, First Monday, Vol. 5, Iss. 10, 2000.

[2] Apple Documentation Archive, Apple, 2018. Available (accessed on 29.7.2018): https://developer.apple.com

[3] AWS, Amazon, 2018. Available (accessed on 25.9.2018): https://aws.amazon.com

[4] Azure, Microsoft, 2018. Available (accessed on 25.9.2018): https://azure.microsoft.com/en-us/

[5] P. Baran, On distributed communications networks, IEEE transactions on Communications Systems, Vol. 12, Iss. 1, 1964, pp. 1–9.

[6] J. Benet, IPFS-content addressed, versioned, P2P file system, arXiv preprint arXiv:1407.3561, 2014.

[7] A. Bori, P. Ekler, The analysis of bittorrent protocol reliability in modern mobile environment, in: 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems, 2013, IEEE, pp. 120–126.

[8] J. Buford, H. Yu, Peer-to-Peer Networking and Applications: Synopsis and Research Directions, in: Shen, X., Yu, H., Buford, J., Akon, M. (eds.), Handbook of Peer-to-Peer Networking, Springer Science & Business Media, 2010, pp. 223–280.

[9] J. Buford, H. Yu, E.K. Lua, P2P networking and applications, Morgan Kaufmann, 2009.

[10] S. Cherbal, A. Boukerram, A. Boubetra, An improvement of mobile Chord protocol using locality awareness on top of cellular networks, in: 5th International Conference on Electrical Engineering-Boumerdes (ICEE-B), 2017, IEEE, pp. 1–6.

[11] F. Chowdhury, J. Furness, M. Kolberg, Performance analysis of structured peer-to-peer overlays for mobile networks, International Journal of Parallel, Emergent and Distributed Systems, Vol. 32, Iss. 5, 2017, pp. 522–548.

[12] Cisco Global Cloud Index: Forecast and Methodology, 2016–2021, Cisco, White Paper, 2018, 46 p. Available (accessed on 7.6.2018): https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf

[13] I. Clarke, O. Sandberg, B. Wiley, T.W. Hong, Freenet A Distributed Anonymous Information Storage and Retrieval System, in: Designing privacy enhancing technologies, 2001, Springer, Berlin, Heidelberg, pp. 46–66.

[14] B. Cohen, Incentives build robustness in BitTorrent, Workshop on Economics of Peer-to-Peer systems, Vol. 6, 2003, pp. 68–72.

[15] K. Csorba, P. Ekler, I. Kelényi, Analyzing Content Life Cycle in Mobile Content Sharing Environment, in: 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC), IEEE, 2011, pp. 92–98.

[16] Dat Project Github Organization, Github, 2018. Available (accessed on 23.8.2018): https://github.com/datproject

[17] P. De Filippi, S. McCarthy, Cloud Computing: Centralization and Data Sovereignty, European Journal for Law and Technology, Vol. 3, Iss. 2, 2012. Available (accessed on 7.6.2018): http://ejlt.org/article/view/101/234

[18] DeviceSpecifications, DeviceSpecifications, 2018. Available (accessed on 20.9.2018): https://www.devicespecifications.com/

[19] K. Dhara, Y. Guo, M. Kolberg, X. Wu, Overview of Structured Peer-to-Peer Overlay Algorithms, in: Shen, X., Yu, H., Buford, J., Akon, M. (eds.), Handbook of Peer-to-Peer Networking, Springer Science & Business Media, 2010, pp. 223–254.

[20] Sailfish OS Platform SDK Containers, Docker Hub, 2018. Available (accessed on 23.8.2018): https://hub.docker.com/r/coderus/sailfishos-platform-sdk-base/

[21] A. Eivy, Be Wary of the Economics of "Serverless" Cloud Computing, IEEE Cloud Computing, Vol. 4, Iss. 2, 2017, pp. 6–12.

[22] P. Ekler, K. Csorba, The Usage and Behavior Patterns of Mobile BitTorrent Clients, Journal of Advanced Computational Intelligence and Intelligent Informatics, Vol. 18, Iss. 3, 2014, pp. 320–323.

[23] P. Ekler, I. Kelényi, I. Dévai, B. Bakos, A. Kiss, Hybrid Peer-to-Peer Content Sharing in Mobile Networks., JNW, Vol. 4, Iss. 2, 2009, pp. 119–132.

[24] P. Ekler, J.K. Nurminen, A. Kiss, Experiences of implementing BitTorrent on Java ME platform, in: Consumer Communications and Networking Conference, 2008, Citeseer, pp. 1154–1158.

[25] Ethereum Project, Ethereum, 2018. Available (accessed on 25.8.2018): https://www.ethereum.org/

[26] Ethereum Github Organization, Github, 2018. Available (accessed on 23.8.2018): https://github.com/ethereum

[27] G. Fettweis, S. Alamouti, 5G: Personal mobile internet beyond what cellular did to telephony, IEEE Communications Magazine, Vol. 52, Iss. 2, 2014, pp. 140–145.

[28] Git, Git, 2018. Available (accessed on 25.8.2018): https://git-scm.com/

[29] The Go Programming Language, Google, 2018. Available (accessed on 15.8.2018): https://golang.org/

[30] go-ipfs, Github, 2018. Available (accessed on 25.8.2018): https://github.com/ipfs/go-ipfs

[31] Android Developers Documentation, Google, 2018. Available (accessed on 29.7.2018): https://developer.android.com/docs/

[32] GSMArena, GSMArena, 2018. Available (accessed on 20.9.2018): https://www.gsmarena.com/

[33] S. Gurun, P. Nagpurkar, B.Y. Zhao, Energy consumption and conservation in mobile peer-to-peer systems, in: Proceedings of the 1st international workshop on Decentralized resource sharing in mobile computing and networking, ACM, 2006, pp. 18–23.

[34] M. Halpern, Y. Zhu, V.J. Reddi, Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction, in: International Symposium on High Performance Computer Architecture (HPCA), March 12–16, 2016, IEEE, Barcelona, Spain, pp. 64–76.

[35] M.V. Heikkinen, A. Kivi, H. Verkasalo, Measuring mobile peer-to-peer usage: Case Finland 2007, in: International Conference on Passive and Active Network Measurement, 2009, Springer, pp. 165–174.

[36] O.P. Heinisuo, V. Lenarduzzi, D. Taibi, Asterism: Decentralized File Sharing Application for Mobile Devices, in: The Seventh IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, April 4–9, 2019, San Francisco East Bay, California, USA. (Submitted on 15.11.2018.).

[37] IPFS Github Organization, Github, 2018. Available (accessed on 23.8.2018): https://github.com/ipfs/

[38] IPFSDroid, Github, 2018. Available (accessed on 29.7.2018): https://github.com/ligi/IPFSDroid

[39] IPLD, Protocol Labs, 2018. Available (accessed on 20.8.2018): https://ipld.io/

[40] M.E. Johnson, D. McGuire, N.D. Willey, Why file sharing networks are dangerous?, Communications of the ACM, Vol. 52, Iss. 2, 2009, pp. 134–138.

[41] S. Kaune, R.C. Rumin, G. Tyson, A. Mauthe, C. Guerrero, R. Steinmetz, Unraveling BitTorrent's File Unavailability: Measurements, Analysis and Solution Exploration, Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference, 2009, pp. 1–9.

[42] I. Kelényi, Á. Ludányi, J.K. Nurminen, Distributed BitTorrent proxy for energy efficient mobile content sharing, in: International Symposium on Wireless Personal Multimedia Communications (WPMC), 2011, IEEE, pp. 1–5.

[43] I. Kelényi, J.K. Nurminen, Á. Ludányi, T. Lukovszki, Modeling resource constrained BitTorrent proxies for energy efficient mobile content sharing, Peer-to-Peer Networking and Applications, Vol. 5, Iss. 2, 2012, pp. 163–177.

[44] M.A. Khan, L. Yeh, K. Zeitouni, C. Borcea, MobiStore: A system for efficient mobile P2P data sharing, Peer-to-Peer Networking and Applications, Vol. 10, Iss. 4, 2017, pp. 910–924.

[45] libp2p, Protocol Labs, 2018. Available (accessed on 20.8.2018): https://libp2p.io/

[46] L. Liu, N. Antonopoulos, From Client-Server to P2P Networking, in: Shen, X., Yu, H., Buford, J., Akon, M. (eds.), Handbook of Peer-to-Peer Networking, Springer Science & Business Media, 2010, pp. 72–89.

[47] T. Locher, P. Moor, S. Schmid, R. Wattenhofer, Free Riding in BitTorrent is Cheap, 2006.

[48] M. Matuszewski, N. Beijar, J. Lehtinen, T. Hyyrylainen, Understanding attitudes towards mobile peer-to-peer content sharing services, in: IEEE International Conference on Portable Information Devices, 2007, IEEE, pp. 1–5.

[49] P. Maymounkov, D. Mazieres, Kademlia: A peer-to-peer information system based on the xor metric, in: International Workshop on Peer-to-Peer Systems, 2002, Springer, pp. 53–65.

[50] D.D.F. Mazières, Self-certifying file system, dissertation, Massachusetts Institute of Technology, 2000.

[51] Nethogs, Github, 2018. Available (accessed on 26.9.2018): https://github.com/raboof/nethogs

[52] S. Nichols, AWS's S3 outage was so bad Amazon couldn't get into its own dashboard to warn the world, The Register, 2017. Available (accessed on 21.6.2018): https://www.theregister.co.uk/2017/03/01/aws_s3_outage/

[53] J. Nurminen, J. Nöyränen, Energy-consumption in mobile peer-to-peer-quantitative results from file sharing, in: Consumer Communications and Networking Conference, 2008, IEEE, Las Vegas, NV, USA, pp. 729–733.

[54] M. Ogden, Dat - Distributed Dataset Synchronization And Versioning, 2017.

[55] S. Ratnasamy, A scalable content-addressable network, dissertation, University of California at Berkeley, 2002.

[56] J. Ritter, Why Gnutella Can't Scale. No, really., 2001. Available (accessed on 5.7.2018): http://www.cs.rice.edu/~alc/old/comp520/papers/ritter01gnutella-cant-scale.pdf

[57] M. Rosenberg, N. Confessore, C. Cadwalladr, How Trump Consultants Exploited the Facebook Data of Millions, New York Times, 2018. Available (accessed on 7.6.2018): https://www.nytimes.com/2018/03/17/us/politics/cambridge-analytica-trump-campaign.html

[58] A. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems, in: IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, 2001, Springer, Berlin, Heidelberg, pp. 329–350.

[59] R. Schollmeier, A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications, in: Proceedings First International Conference on Peer-to-Peer Computing, August 27–29, 2001, IEEE, Linkoping, Sweden, pp. 101–102.

[60] Sailfish OS, Jolla, 2018. Available (accessed on 20.7.2018): https://sailfishos.org/

[61] Mobile and tablet internet usage exceeds desktop for first time worldwide, Statcounter, 2016. Available (accessed on 3.6.2018): http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide

[62] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M.F. Kaashoek, F. Dabek, H. Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, ACM SIGCOMM Computer Communication Review, Vol. 31, Iss. 4, 2001.

[63] Swarm, Swarm, 2018. Available (accessed on 15.7.2018): http://swarm-gateways.net/bzz:/theswarm.eth/

[64] Travis CI, Travis CI, 2018. Available (accessed on 23.8.2018): https://travis-ci.org

[65] H. Zhang, Y. Wen, H. Xie, N. Yo, Distributed hash table: Theory, platforms and applications, Springer, 2013.

[66] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, J.D. Kubiatowicz, Tapestry: A resilient global-scale overlay for service deployment, IEEE Journal on selected areas in communications, Vol. 22, Iss. 1, 2004, pp. 41–53.